

COURSE TEXTBOOK

Object-Oriented Programming with Java

Sercan Külcü

Giresun University · Department of Computer Engineering

Object-Oriented Programming with Java

Sercan Külcü
Giresun University, Department of Computer Engineering
Giresun, Türkiye

This book is the printed form of the course notes published at
`sercankulcu.github.io/teaching/object-oriented-programming-with-java/`

Slides, study questions and simulators are on that page. The code is in the `github.com/sercankulcu/object-oriented-programming-java-en` repository.

© Sercan Külcü. All rights reserved.

Generated from the notes of 27 November 2026.

Preface

This book starts from the foundations of programming and leads to object-oriented programming and real applications in Java: classes and objects, encapsulation, inheritance, polymorphism and abstraction; collections, exceptions, testing, file input and output, graphical user interfaces, threads and database programming. It is made up of the notes for the Object-Oriented Programming with Java course taught in the Department of Computer Engineering at Giresun University, and its chapters follow the order of the course.

The book has four parts. In order, they are **Programming Fundamentals** (Chapters 1–3), **Object-Oriented Programming** (Chapters 4–5), **Robust Programs** (Chapters 6–7), and **Applications in Java** (Chapters 8–11).

Some chapters are made up of more than one set of lecture notes. These chapters open with a short title page, and the notes follow as numbered sections such as 4.1 and 4.2.

Every chapter follows the same layout. It opens with learning outcomes. Formal definitions and worked examples are set in named boxes. *Common mistake* boxes point out the misconceptions students fall into most often. Code traces show, step by step, how a program runs. Each chapter closes with a summary, a list of key terms, *Test Yourself* questions with their answers, suggestions for further reading and a bibliography. The glossary at the end of the book defines each key term in a sentence and names the chapter that explains it.

The code for every chapter is in the companion repository `github.com/sercankulcu/object-oriented-programming-java-en`, where each program is compiled, run and checked against its expected output automatically. Slides, study questions and simulators are on the course page.

Sercan Külcü
Giresun

Contents

Course Map 1

PART I · PROGRAMMING FUNDAMENTALS

1 Computers, Computation and Algorithms 4

1.1 Computers 5
1.2 Computation and Computational Thinking 52
1.3 Algorithms 104
1.4 Flowcharts 158

2 Programming and Java Syntax 210

2.1 Programming 211
2.2 Syntax Rules 261

3 Control Structures, Methods, Arrays and Strings 324

3.1 Control Structures 325
3.2 Methods 385
3.3 Arrays 451
3.4 Strings 525
3.5 Recursion 577

PART II · OBJECT-ORIENTED PROGRAMMING

4 Objects and Classes, Inheritance and Polymorphism 628

4.1 Objects and Classes 629
4.2 Encapsulation 715
4.3 Inheritance 786
4.4 Polymorphism 871
4.5 Abstract Classes and Interfaces 934

5 Collections 1029

5.1 The Collections Framework 1030
5.2 Lists 1040
5.3 Queues 1055
5.4 Sets 1066
5.5 Maps 1078
5.6 The Iterator 1087
5.7 Algorithms 1092
5.8 The Stream API 1102
5.9 Choosing the right collection 1111
5.10 Solved problems 1112
5.11 Common mistakes 1118
5.12 Summary 1121

5.13 Key terms 1122
5.14 Test Yourself 1122
5.15 Further reading 1124
5.16 References 1124

PART III · ROBUST PROGRAMS

6 Exceptions 1127
6.1 The idea of an exception 1128
6.2 try and catch 1133
6.3 finally and resource management 1142
6.4 The exception hierarchy 1152
6.5 Checked and unchecked exceptions 1154
6.6 Throwing exceptions 1158
6.7 Runtime exceptions 1167
6.8 Checked exception classes 1178
6.9 Error classes 1183
6.10 Custom exceptions 1185
6.11 Stack trace and good practices 1191
6.12 Solved problems 1196
6.13 Common mistakes 1201
6.14 Summary 1202
6.15 Key terms 1203
6.16 Test Yourself 1204
6.17 Further reading 1206
6.18 References 1206

7 Testing and Debugging 1207
7.1 The foundations of testing and debugging 1208
7.2 Test levels 1210
7.3 Unit testing and JUnit 1213
7.4 Good test design 1230
7.5 Regression and automation 1236
7.6 Debugging 1241
7.7 Analysis tools 1250
7.8 Solved problems 1265
7.9 Common mistakes 1270
7.10 Summary 1271
7.11 Key terms 1272
7.12 Test Yourself 1272
7.13 Further reading 1274
7.14 References 1274

PART IV · APPLICATIONS IN JAVA

8 Input and Output 1277

8.1 Streams 1278
8.2 Files and encodings 1284
8.3 The file system: the File class 1293
8.4 Reading and writing text 1296
8.5 Buffering 1311
8.6 Other byte streams 1317
8.7 Copying files 1323
8.8 Random access 1333
8.9 Serialization 1338
8.10 NIO 1344
8.11 Solved problems and exercises 1359
8.12 Common mistakes 1362
8.13 Summary 1363
8.14 Key terms 1364
8.15 Test Yourself 1365
8.16 Further reading 1366
8.17 References 1367

9 Graphics and User Interfaces 1368

9.1 Graphics programming 1369
9.2 Windows and panels 1379
9.3 Drawing tools 1383
9.4 Layout managers 1389
9.5 Application: the no-entry sign 1396
9.6 Mouse events and the event dispatch thread 1402
9.7 Application: a drawing program 1410
9.8 Buttons and text components 1418
9.9 Application: a text converter 1424
9.10 Application: a change calculator 1427
9.11 Best practices 1433
9.12 Solved problems 1434
9.13 Common mistakes 1436
9.14 Summary 1438
9.15 Key terms 1440
9.16 Test Yourself 1440
9.17 Further reading 1442
9.18 References 1442

10 Threads 1444

10.1 Basic concepts 1445
10.2 The Thread class 1450
10.3 The life cycle 1454
10.4 Creating and starting threads 1457
10.5 Managing and stopping threads 1467
10.6 Synchronization 1473
10.7 Visibility 1482

10.8 Explicit locks and deadlock 1486
10.9 Condition synchronization 1500
10.10 Synchronizer objects 1510
10.11 Other tools 1517
10.12 Worked problems 1530
10.13 Common mistakes 1533
10.14 Summary 1534
10.15 Key terms 1535
10.16 Test Yourself 1536
10.17 Further reading 1538
10.18 References 1538

11 Database Programming 1539

11.1 The idea of a database 1540
11.2 SQLite 1545
11.3 SQL basics 1547
11.4 Relationships and grouping 1555
11.5 JDBC 1568
11.6 Basic operations 1574
11.7 PreparedStatement 1583
11.8 Advanced JDBC 1591
11.9 Transactions 1597
11.10 Worked examples and exercises 1604
11.11 Common mistakes 1607
11.12 Summary 1609
11.13 Key terms 1610
11.14 Test Yourself 1610
11.15 Further reading 1612
11.16 References 1612

APPENDICES

A Glossary of Terms 1613

B Prerequisites for Object-Oriented Programming with Java 1617

C Important People in Object-Oriented Programming with Java 1637

Course Map

This page puts every chapter of the course on one map. Each chapter is a node, and an arrow shows which idea from an earlier topic the next one relies on. Read it before the course starts to see where the topics are heading, or during the course to see where you are.

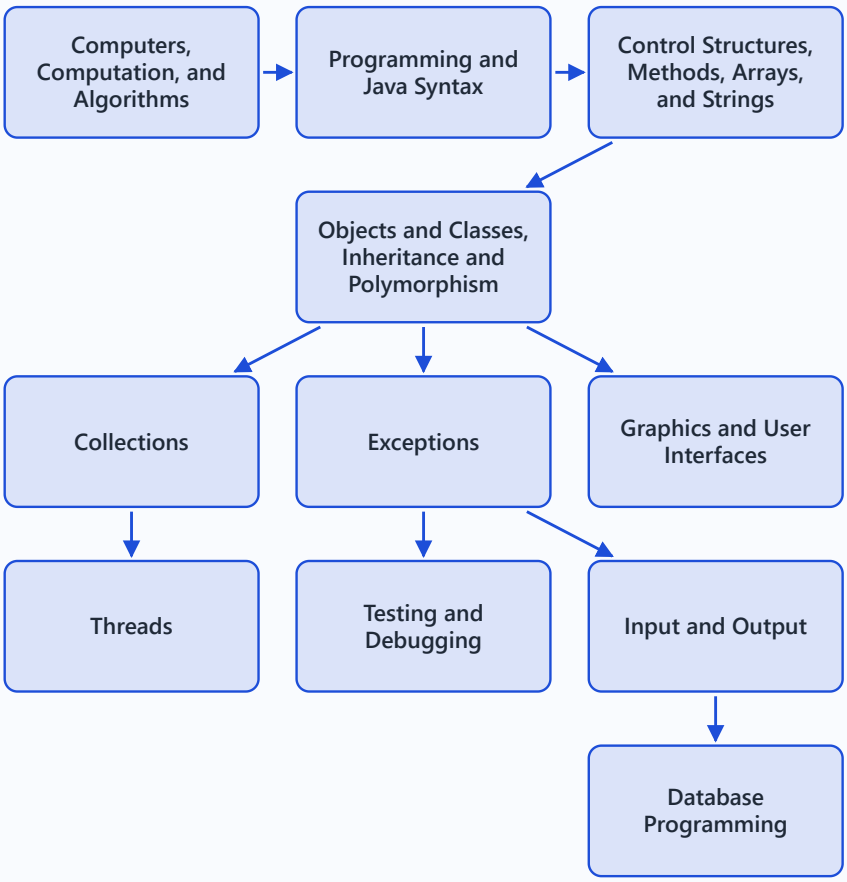


Figure 1: The course map. Each arrow points from a topic to one that builds on it. The programming basics lead to objects, and objects lead to robust programs and applications.

Figure 1 shows the connections between the topics. Each node is a chapter, and an arrow indicates which idea of the earlier topic the later one relies on. Only the closest connections are drawn on the map; every chapter that a given chapter depends on is listed in Table 1.

Topic (chapter)	Builds on	Connection
Computers, Computation, and Algorithms (1)	—	Representing information in memory, the stored program, and the algorithm; the ground for programming.
Programming and Java Syntax (2)	Computers, Computation, and Algorithms	The syntax and meaning of a programming language; compilation, types, and variables in Java.
Control Structures, Methods, Arrays, and Strings (3)	Programming and Java Syntax	Structures that direct the flow, methods, arrays, and strings; the last tools before objects.
Objects and Classes, Inheritance, and Polymorphism (4)	Control Structures, Methods, Arrays, and Strings	Data and behavior come together in the class: encapsulation, inheritance, polymorphism, interfaces.
Collections (5)	Objects and Classes, Inheritance, and Polymorphism	Ready-made classes and interfaces: List, Set, Map; generics and comparators.
Exceptions (6)	Objects and Classes, Inheritance, and Polymorphism	An error is an object too: the exception hierarchy is built with inheritance; try, catch, and finally.
Testing and Debugging (7)	Objects and Classes, Inheritance, and Polymorphism, Exceptions	Testing the behavior of classes with JUnit; testing an expected exception; debugging.
Input and Output (8)	Collections, Exceptions	Streams and files; I/O errors are reported with checked exceptions.
Graphics and User Interfaces (9)	Objects and Classes, Inheritance, and Polymorphism	Swing components are extended through inheritance; events are caught with listener interfaces.
Threads (10)	Objects and Classes, Inheritance, and Polymorphism, Collections	The Runnable interface and the Thread class; race conditions in shared collections.
Database Programming (11)	Exceptions, Input and Output	The second form of persistent data: from files to databases; JDBC and SQLException.

Table 1: The connections on the map.

The backbone of the map can be read as **foundations → object → robust program → application**. The first three chapters set up the language and structured programming. The chapter on objects and classes is the center of the course: every later chapter uses one of its tools. Collections use interfaces and generics, exceptions use the inheritance hierarchy, and graphical interfaces use inheritance and listener interfaces. Testing and exceptions deal with the robustness of the program, while input/output and databases deal with the persistence of data.

PART I

Programming Fundamentals

Chapters 1–3

PART I · CHAPTER 1

Computers, Computation and Algorithms

Programming rests on how a computer represents and processes information. The chapter opens with the representation of numbers, characters and strings in memory, the stored-program idea and the von Neumann architecture. It then touches on the history of computer science, Turing machines and the limits of computation, and approaches problem solving through the notion of an algorithm with simple searching and sorting examples. The last part introduces flowchart symbols and works through sequential, conditional and repeated flow on sample problems.

NOTES IN THIS CHAPTER

1.1 Computers

1.2 Computation and Computational Thinking

1.3 Algorithms

1.4 Flowcharts

PART I · CHAPTER 1.1

Computers

Every line of Java you write ends up as numbers that a processor understands. Why can an `int` never exceed 2,147,483,647? Why does `"😀".length()` return 2, and why is `0.1 + 0.2` not exactly 0.3? The answers all lie in that translation. These notes build the computer layer by layer. First, electrical signals become bits and bits become numbers. Numbers stand for letters and text, and sequences of numbers stand for images, sound and video. Last comes the stored-program idea, in which the instructions themselves sit in memory as numbers, and the von Neumann architecture that puts it into hardware. At every layer we show what the idea looks like in Java, using short programs whose output we trace step by step.

LEARNING OUTCOMES

Hardware
Explain the roles of the processor, memory, disk and bus, and trace step by step how a program runs on the processor through the fetch–decode–execute cycle.

Numbers
Convert numbers between binary, octal, hexadecimal and decimal; add binary numbers; and compute with negative numbers in two's complement.

Text
Explain the difference between ASCII, Unicode and UTF-8, and show how a string is represented in memory as numbers.

Connection to Java
Explain the limits of `byte` and `int` and overflow through two's complement; work with number bases in Java; relate `char` and `String` length to UTF-16; and show how assembly instructions map to machine code.

1.1.1 Computer basics

A computer is a machine that carries out given instructions very fast and without error, one after another. Each instruction is very simple on its own: add two numbers, read a number from memory, write the result to memory, jump to another instruction if a condition holds. What makes a computer

powerful is that it can do these simple steps billions of times a second. The classes, objects and methods of an object-oriented program are in the end reduced to these same simple steps, so starting Java from the machine itself explains a good deal of behavior you will meet later.

1.1.1.1 Basic components

A computer contains a great many components, but three stand out from a programmer's point of view. They are connected by paths that carry data between them.

- **Processor** (CPU, Central Processing Unit): regarded as the computer's "brain". It performs all arithmetic and logical computation: addition, subtraction, comparing two numbers and logic operations on bits are its job.
- **Memory** (RAM, Random Access Memory): temporarily stores the running program and its data. "Random access" means any cell can be reached in the same amount of time, without stepping through the others in order. The capacity and access speed of memory directly affect the speed of the computer.
- **Disk** (HDD or SSD): stores data permanently. When the power is cut, everything in RAM is erased, while the files on disk stay where they are. An HDD uses spinning magnetic platters, whereas an SSD uses semiconductor memory chips with no moving parts; both are much larger and much slower than RAM.

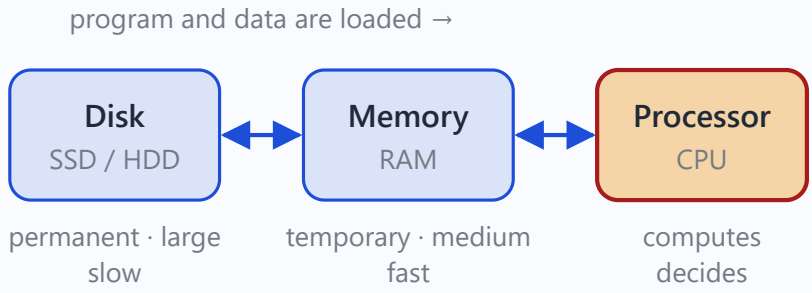


Figure 1.1.1: The flow of data between disk, memory and processor. The program and its data are loaded from disk into memory, and the processor computes by accessing memory. The disk is permanent but slow; memory is temporary but fast.

A kitchen analogy makes the relationship among the three easy to remember. The cook is the processor. The counter is RAM: small, but right within the cook's reach. The pantry is the disk: very large, but far away. The cook does not cook directly from the pantry. First the cook carries the needed ingredients from the pantry to the counter and works there. Likewise, when a program is started, its code and data are first loaded from disk into memory, and the processor works only with the instructions and data in memory. If the counter fills up, the cook keeps walking to the pantry and the cooking slows down; a computer slows down for the same reason when RAM runs short.

From an electronics point of view, the difference between RAM and disk comes from how information is held. The widely used dynamic RAM (DRAM) stores each bit as a charge on a tiny capacitor. That charge leaks away by itself, so it must be refreshed many times a second, and the information is lost when the power goes off. Flash memory in an SSD, by contrast, traps the charge inside an insulated gate, and the charge stays there for years even without power. This is why RAM is called **volatile** and disk **non-volatile** memory.

1.1.1.2 How a computer works, and programming

How a computer works can be summed up in three basic steps. Instructions are sent to the processor. The processor carries them out in order. The results are presented to the user or to other software. Printing text on the screen, saving a file and sending a network packet are all products of this cycle.

> **Definition 1.1.1**
>
> **Programming** is the process of writing instructions for a computer, in a form it can understand, so that it performs a desired task. The whole body of instructions written is called a **program**.

The shortest Java program is one that writes a single line to the screen. It contains just one instruction: "print to the screen".

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

```
Hello, World!
```

The boilerplate of the program can look crowded at first. The line `public class HelloWorld` defines a class; in Java all code lives inside a class, and the file must be named `HelloWorld.java`. The `main` method is where the program starts; when the Java Virtual Machine (JVM) runs a program, it calls this method first. The real work is in the line `System.out.println(...)`, which writes the quoted string to standard output and ends the line.

The processor does not know the word `println`. Instructions written in a language like Java are run by translating them into **machine code**, which the processor understands. In Java this translation has two stages. First the `javac` compiler translates the source code into an intermediate code called **bytecode**. Then the JVM interprets this code, or compiles its frequently executed parts into the

machine code of that processor. In the end, even drawing a letter on the screen amounts to the processor moving numbers from one place to another. The rest of these notes explains what those numbers represent and how.

1.1.1.3 A computer in layers

It is hard to grasp how a computer works all at once. It is easier to build the computer up layer by layer, each layer resting on the one below and using the building blocks it provides.

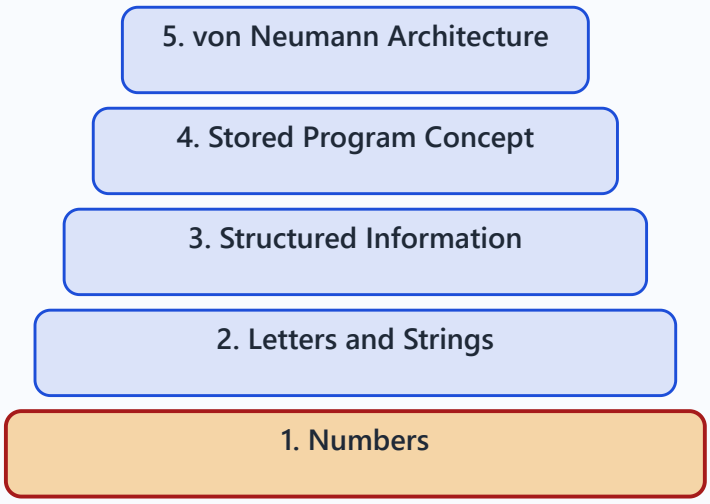


Figure 1.1.2: A road map of these notes. Numbers are at the bottom; letters are built from numbers, and structured information from letters. Instructions are also stored as numbers, and the von Neumann architecture brings everything together.

1. First, electrical signals are turned into numbers.
2. Numbers represent letters, and letters represent text.
3. Sequences of numbers store structured information such as images, sound and video.
4. Instructions are also stored in memory as numbers.
5. The von Neumann architecture brings it all together.

This order also determines the sections of these notes. At each layer we ask the same question: how is a concept from the layer above expressed with the numbers below it? The answers also explain why Java's primitive types (`byte` , `int` , `char` , `double`) are designed the way they are.

1.1.2 Binary numbers

1.1.2.1 From electrical signal to bit

The basic job of a computer is to process electrical signals according to fixed rules. These rules give rise to functions such as addition, comparison and copying. A signal carries no meaning on its own. Only when we associate signals with numbers and symbols that people understand can the computer present us with meaningful information.

Computers process signals using a **binary** (base 2) representation. If the voltage on a wire is above a certain threshold, it is read as 1; if below, as 0. All operations are built on this binary foundation.

> **Definition 1.1.2**
>
> A **bit** (binary digit) is the smallest unit of information, able to take only the value 0 or 1. A group of eight bits is called a **byte**.

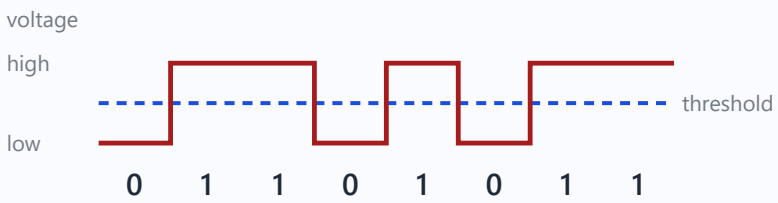


Figure 1.1.3: The voltage on a wire varies over time. Intervals above the threshold are read as 1 and those below as 0; this wave carries the bit sequence 01101011.

The values 0 and 1 go by different names depending on context. Table 1.1.1 shows the most common ones. In logic, 0 is false and 1 is true; on a switch, 0 is off and 1 is on; in a circuit, 0 is low voltage and 1 is high voltage. They are all different readings of the same duality.

0	1
False (`false`)	True (`true`)
Off	On
Low voltage	High voltage

Table 1.1.1: Different readings of the two values of a bit.

Why two states rather than ten? On a real wire the voltage never sits exactly at the intended value. Heat, interference from neighboring wires and ripple in the power supply add **noise** to the signal. It is easy to tell whether a voltage is "present or absent" despite noise, because a wide safety margin can be left between the two states. Dividing the same voltage range into ten equal levels, on the other

hand, narrows every level. A little noise pushes the signal into the neighboring level, and a digit sent as 7 is read as 6. This is why logic gates in electronics work with two states, and digital circuits are extremely reliable thanks to that safety margin.

Memory and file sizes are measured in bytes. Prefixes are used in two different ways. In the International System of Units (SI), 1 kilobyte (KB) is 1000 bytes. In the binary prefixes based on powers of two, 1 kibibyte (KiB) is 1024 bytes. The gap grows as the units get larger: 1 GB = 10^9 bytes, while 1 GiB = 2^{30} = 1,073,741,824 bytes. A disk labeled 1 TB shows up as roughly 931 "GB" in the operating system because the operating system actually computes in GiB.

1.1.2.2 Place value

The decimal system we use every day is a **positional** system. In the number 603, the 6 is worth 600 because it sits in the hundreds place. The same digit carries a different value in a different position. The binary system works on the same principle, except that the base is 2 instead of 10.

An integer is written as a sum of powers of 2, where each power is either not used (0) or used once (1). Four consecutive powers, 2^0, 2^1, 2^2, 2^3, represent all integers from 0 to 15. The number 13 is written as follows:

$$13_{10} = 1\cdot2^3 + 1\cdot2^2 + 0\cdot2^1 + 1\cdot2^0 = 8 + 4 + 0 + 1 = 1101_2$$

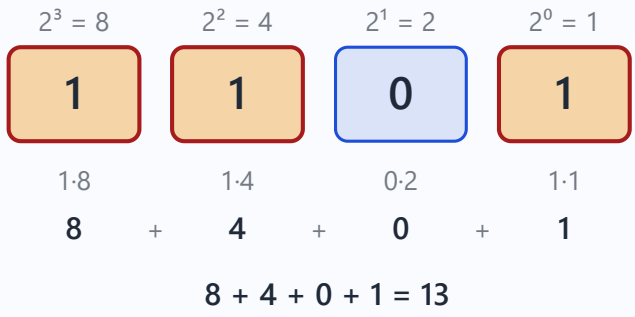


Figure 1.1.4: In 1101_2, the place weights from left to right are 8, 4, 2 and 1. The weights of the places holding 1 are added: 8 + 4 + 0 + 1 = 13.

The small digit at the lower right of the number, the **subscript**, shows the base. 1101_2 is read "one one zero one, base two", not "one thousand one hundred one". Numbers without a subscript are taken to be decimal. The same rule holds in the other bases (Table 1.1.2).

Base	Example
Decimal (10)	$603_{10} = 6\cdot10^2 + 0\cdot10^1 + 3\cdot10^0$
Octal (8)	$207_8 = 2\cdot8^2 + 0\cdot8^1 + 7\cdot8^0 = 135_{10}$

Base	Example
Hexadecimal (16)	$FF_{16} = 15 \cdot 16^1 + 15 \cdot 16^0 = 255_{10}$

Table 1.1.2: The place-value rule is the same in every base.

Converting a binary number to decimal is therefore easy: add up the weights of the places that hold 1. In 10110110_2, the 1s are in places 7, 5, 4, 2 and 1 (the rightmost place is place 0). The value is $128 + 32 + 16 + 4 + 2 = 182$. Knowing the weights by heart speeds this up: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024. Programmers know this sequence like a multiplication table.

1.1.2.3 Any base, and bit limits

The idea of place value generalizes to every base. If the base is $b \geq 2$, each digit takes one of the values $0, 1, \ldots, b - 1$. In binary the digits are 0–1, in octal 0–7, and in hexadecimal 0–9 and A–F, where A = 10, B = 11, C = 12, D = 13, E = 14 and F = 15. To count up to 15 with a single digit, six new symbols are needed.

> **Theorem 1.1.1 (Digit limit)**
>
> With base $b \geq 2$ and d digits, every integer from 0 to $b^d - 1$ can be written, and no other integer can.

Proof. Each digit can take b different values, so d digits give b^d different digit strings in total. The smallest string is 00…0 = 0. In the largest string every digit is $b - 1$; adding 1 to it carries all the way and opens a new digit on the far left, giving b^d. So the largest value is $b^d - 1$. There are b^d numbers in between, and each corresponds to a different string. ■

In decimal, three digits write 0–999: $10^3 - 1 = 999$. In binary, the largest unsigned value with d bits is $2^d - 1$ (Table 1.1.3). A 64-bit computer can count, unsigned, up to $2^{64} - 1$ = 18,446,744,073,709,551,615.

Bits	Largest ($2^d - 1$)
4	15
8	255
16	65.535
32	4.294.967.295

Table 1.1.3: The largest unsigned value by number of bits.

Java's integer types, however, are **signed**: half of the values the bits can represent are set aside for negative numbers. That is why the 8-bit `byte` goes up to 127, not 255. The following program prints the largest value of the four integer types.

```
System.out.println(Byte.MAX_VALUE);    // 8 bit
System.out.println(Short.MAX_VALUE);   // 16 bit
System.out.println(Integer.MAX_VALUE); // 32 bit
System.out.println(Long.MAX_VALUE);    // 64 bit
// all 64 bits 1, read unsigned:
System.out.println(Long.toUnsignedString(-1L));
```

```
127
32767
2147483647
9223372036854775807
18446744073709551615
```

Each constant is about half the unsigned limit in the table. `Byte.MAX_VALUE` = 127 = $2^7 - 1$, `Short.MAX_VALUE` = 32,767 = $2^{15} - 1$, `Integer.MAX_VALUE` = 2,147,483,647 = $2^{31} - 1$ and `Long.MAX_VALUE` = $2^{63} - 1$. One bit is evidently used to tell negative numbers apart. All 64 bits of `-1L` are 1. When `Long.toUnsignedString` reads those bits as unsigned, it returns $2^{64} - 1$. The same bits can be −1 or a number larger than 18 quintillion, depending on how they are interpreted. This duality returns when we get to negative numbers.

In Java, `int` is 32 bits on every platform and `long` is 64 bits on every platform. In C, the size of `int` can vary with the compiler and the processor. Java's fixed sizes ensure that the same program gives the same result on every machine.

1.1.2.4 Decimal to binary: repeated division by 2

The most systematic way to convert a decimal number to binary is to divide it by 2 repeatedly. At each division the remainder is 0 or 1, and this remainder is the rightmost bit of the number not yet written down. Stop when the quotient is 0 and read the remainders from bottom to top.

To see why the method works, it is enough to look at how a number is written in binary. If $13 = 1101_2$, then $13 = 2 \cdot 6 + 1$. The remainder 1 is the rightmost bit, and the quotient 6 is the number formed by the remaining bits, namely 110_2. Applying the same step to 6 produces the next bit. Each division "snips off" one bit and shifts the number one bit to the right. The following program implements this method in Java.

```
public static void main(String[] args) {
  int n = 13;
  String bits = "";
  while (n > 0) {
    int rem = n % 2;
    System.out.println(n + " % 2 = " + rem);
    bits = rem + bits;
    n = n / 2;
  }
  System.out.println("Result: " + bits);
}
```

Four points in the code matter. `n % 2` is the remainder (modulo) operation and yields the last bit. `n / 2` is **integer division**, because it is between two `int`s: the fraction is discarded, so 13 / 2 = 6. In the line `bits = rem + bits`, `rem` is an `int` and `bits` is a `String`. A number added to a string is converted to a string, and the two strings are concatenated. Because each new bit is added to the *front* of the string, the bits accumulate in the right order and the remainders do not need to be reversed separately. The loop ends when `n` reaches 0. Table 1.1.4 shows each iteration for 13.

Iteration	Condition	`rem`	Line printed	`bits`	New `n`
1	13 > 0 true	1	`13 % 2 = 1`	`"1"`	6
2	6 > 0 true	0	`6 % 2 = 0`	`"01"`	3
3	3 > 0 true	1	`3 % 2 = 1`	`"101"`	1
4	1 > 0 true	1	`1 % 2 = 1`	`"1101"`	0
—	0 > 0 false	The loop ends; `Result: 1101` is printed.			

Table 1.1.4: Trace of the loop for `n = 13`.

```
13 % 2 = 1
6 % 2 = 0
3 % 2 = 1
1 % 2 = 1
Result: 1101
```



Figure 1.1.5: Dividing 13 by 2 repeatedly. Each row shows the number, its half and the remainder. Reading the remainders from bottom to top gives 1101.

The loop ran four times because the binary form of 13 has four bits. In general, for $n \geq 1$ the number of iterations is $\lfloor \log_2 n \rfloor + 1$: the number is halved each time, and this is how many halvings it takes to fall below 1. The number 1000 is converted in 10 iterations, and one billion in 30.

The code has one limitation: for `n = 0` the loop never runs and `bits` stays empty, so only `Result:` is printed. For negative numbers too, the condition is false on the first iteration. A real program needs a separate case for 0. Java's built-in `Integer.toBinaryString` method handles these details itself.

Example 1.1.1

How is 45 written as an 8-bit binary number? The chain of divisions is: 45 / 2 = 22 remainder 1; 22 / 2 = 11 remainder 0; 11 / 2 = 5 remainder 1; 5 / 2 = 2 remainder 1; 2 / 2 = 1 remainder 0; 1 / 2 = 0 remainder 1. Reading the remainders from bottom to top gives 101101. To fill out eight bits, two zeros are added on the left: **00101101**. Check: $32 + 8 + 4 + 1 = 45$.

The same result can be reached by subtracting powers. The largest power that fits in 45 is 32, leaving 13. The largest power that fits in 13 is 8, leaving 5. Then 4, leaving 1; then 1, leaving 0. The powers used are 32, 8, 4 and 1; those places get a 1 and the others a 0.

1.1.2.5 Octal and hexadecimal

Binary numbers are long. Writing a 32-bit `int` with thirty-two digits is both tiring and error-prone. Programmers therefore use octal and hexadecimal to write binary numbers more compactly. These two bases are chosen because 8 and 16 are powers of 2: $8 = 2^3$ and $16 = 2^4$.

This relationship makes conversion a matter of regrouping rather than arithmetic. One hexadecimal digit corresponds to exactly four bits, and one octal digit to exactly three. To convert from binary to hexadecimal, split the bits into groups of four starting from the right and convert each group to a single hexadecimal digit. For octal the groups have three bits.

Example 1.1.2

Let us convert 10111110_2 to hexadecimal and octal.

For hexadecimal, groups of four: `1011 1110`. 1011 = 8 + 2 + 1 = 11 = B, and 1110 = 8 + 4 + 2 = 14 = E. The result is $\mathbf{BE}_{16}$.

For octal, groups of three from the right: `10 111 110`. The incomplete leftmost group is padded with a zero to give `010`. 010 = 2, 111 = 7, 110 = 6; the result is $\mathbf{276}_{8}$.

Check: *11·16 + 14 = 190* and *2·64 + 7·8 + 6 = 190*. The decimal value of the binary number is also *128 + 32 + 16 + 8 + 4 + 2 = 190*.

Today hexadecimal is far more common than octal. Since a byte is eight bits, it fits exactly two hexadecimal digits: 00–FF. Memory addresses, color codes (`#FF0000` is red), the raw contents of files and Unicode code points are all written in hexadecimal. Octal survives in only a few places, such as Unix file permissions (`chmod 755`).

1.1.2.6 Base conversions in Java

Java provides ready-made methods for converting between bases, and prefixes for writing integer literals in different bases. The following program uses both.

```
System.out.println(Integer.toBinaryString(13));  // binary
System.out.println(Integer.toOctalString(135));  // octal
System.out.println(Integer.toHexString(255));    // hex
System.out.println(Integer.parseInt("1101", 2)); // 2 → 10
System.out.println(Integer.parseInt("207", 8));  // 8 → 10
System.out.println(0b1101 + " " + 0207 + " " + 0xFF);
```

```
1101
207
ff
13
135
13 135 255
```

The first three lines convert an `int` value to a string. `Integer.toBinaryString(13)` gives the binary form, `toOctalString(135)` the octal form and `toHexString(255)` the hexadecimal form. Note that `toHexString` uses lowercase letters: the output is `ff`, not `FF`. `Integer.parseInt` does the conversion in the opposite direction. Its second argument says which base the string is written in. `parseInt("1101", 2)` reads the string as binary and returns 13. The returned value is now an ordinary `int`; the base is a property of the notation, not of the number.

The last line shows how to write literals directly in different bases in source code. These notations are called **literals**, and their prefixes are summarized in Table 1.1.5.

Prefix	Base	Example	Note
`0b` ya da `0B`	Binary	`0b1101` = 13	Introduced in Java 7.
`0`	Octal	`0207` = 135	`010` is 8, not ten.
`0x` ya da `0X`	Hexadecimal	`0xFF` = 255	Common in color and byte values.
(none)	Decimal	`135`	Must not start with 0.

Table 1.1.5: Prefixes of integer literals in Java.

Prefixes change only the notation. `0b1101`, `015`, `0xD` and `13` produce the same `int` value and sit in memory as the same 32 bits. Long literals can be grouped with underscores: `1_000_000` or `0b1011_1110`. The compiler ignores the underscores; readability improves.

Common mistake

Putting a leading zero in front of a number to align it. `int x = 010;` compiles, but since the literal is read as octal, `x` is 8, not 10. `int y = 08;` does not compile at all: there is no digit 8 in octal, and `javac` reports "`error: ';' expected`". The message looks unrelated at first glance; the compiler takes `0` as a complete literal and then sees the following `8` as an unexpected token.

1.1.2.7 Binary addition

Addition follows the same principles in whatever base is used. Numbers are written one under the other by place value and added from right to left. A value that does not fit in one place is passed to the next place as a **carry**. In decimal, when *7 + 8 = 15*, we write 5 and carry 1. In binary, when *1 + 1 = 2 = 10_2*, we write 0 and carry 1. If three 1s are added together with the carry, *3 = 11_2*: write 1 and carry 1.

All cases of binary addition fit in a four-line table: 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, 1 + 1 = 10. Once the carry is included, three bits are added in each column and the result stays between 0 and 3. In electronics this job is done by a **full adder** circuit: three bits go in, and a sum bit and a carry bit come out. Chaining eight full adders gives an 8-bit adder.

Example 1.1.3

Let us add 57 and 28 as 8-bit binary numbers. *57 = 00111001_2* and *28 = 00011100_2*. Addition starts at the rightmost column (Table 1.1.6).

Column	Bit of 57	Bit of 28	Carry in	Sum	Written	Carry out
1	1	0	0	1	1	0

Column	Bit of 57	Bit of 28	Carry in	Sum	Written	Carry out
2	0	0	0	0	0	0
3	0	1	0	1	1	0
4	1	1	0	$2 = 10_2$	0	1
5	1	1	1	$3 = 11_2$	1	1
6	1	0	1	$2 = 10_2$	0	1
7	0	0	1	1	1	0
8	0	0	0	0	0	0

Table 1.1.6: Column-by-column computation of 57 + 28 (column 1 is the rightmost).



Figure 1.1.6: The finished sum 57 + 28. The red 1s in the top row are carries; the carry generated in the fourth column travels on for three columns. The result is $01010101_2 = 85$.

The result $01010101_2 = 64 + 16 + 4 + 1 = 85$ agrees with $57 + 28 = 85$. No carry came out of the eighth column, so the result fit in 8 bits. Had the sum exceeded 255, a carry would have spilled out of the eighth column and the 8-bit result would have been wrong. This is the first example of **overflow**, which we examine in detail together with negative numbers.

1.1.3 Negative numbers and overflow

1.1.3.1 Two's complement

So far only zero and positive integers have been written. The first idea for negative numbers is to reserve the leftmost bit for the sign: 0 for positive, 1 for negative. This representation is called **sign-magnitude**, and it looks natural on paper. But it has two flaws. Zero gets two separate representations (+0 and −0). More importantly, the adder circuit would have to behave differently depending on the signs. This is why computers use another method.

Definition 1.1.3

In **two's complement** representation, the negative of a positive number is found like this: all bits of the number are flipped (0 ↔ 1), and then 1 is added to the result. The intermediate result obtained by only flipping the bits is called the **one's complement**.

In this representation the leftmost bit behaves like a **sign bit**: if it is 0 the number is positive, if 1 it is negative. The range that 8 bits can represent is −128 … 127. In general, with n bits the range is -2^{n-1} … $2^{n-1} - 1$. There is one more number on the negative side, because zero takes up a place on the positive side.

Example 1.1.4

Let us find −3 on an 8-bit computer. The number 3 is `00000011`.

1. All bits are flipped: `00000011` → `11111100` (the one's complement).
2. 1 is added: `11111100` + 1 = `11111101`.

The result `11111101` is the 8-bit two's complement representation of −3.

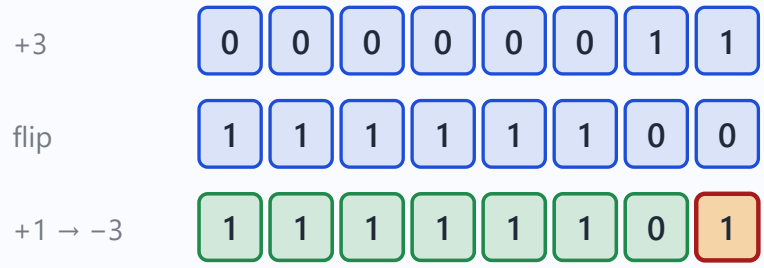


Figure 1.1.7: Finding −3. The top row is 3, the middle row has its bits flipped, and the bottom row is −3 after adding 1. The numbers below are place weights; the weight of the leftmost bit is −128.

The same operation works in the opposite direction. To find the absolute value of a negative number, the bits are again flipped and 1 is added: `11111101` → `00000010` → `00000011` = 3. The operation is its own inverse; applying it twice returns to the start.

Example 1.1.5

The 8-bit representation of −20 is found as follows. $20 = 16 + 4 = 00010100_2$. Flip the bits: `11101011`. Add 1: `11101100`. As a check, reading this bit pattern as unsigned gives $128 + 64 + 32 + 8 + 4 = 236$, and $236 = 256 - 20$. This equality is no coincidence; the next section explains why.

1.1.3.2 Why two's complement works

The main advantage of two's complement is that the adder circuit works unchanged with negative numbers too. Consider the sum $3 + (-3)$:

```
00000011 + 11111101 = 1 00000000
```

The result is nine bits. Since the computer holds only eight bits, the ninth bit on the far left is discarded, leaving exactly `00000000` = 0. The adder circuit never knows the sign of the number; it adds the bits as usual and gives the right result. Subtraction does not need a separate circuit either: $a - b$ is computed as $a + (-b)$. Inverting gates and one adder do both addition and subtraction.

There is a second way to look at the same representation, and it is the shortest way to convert a signed number to decimal. In 8-bit two's complement, the weight of the leftmost bit is not +128 but **−128**. The weights of the other bits do not change. So if the leftmost bit is 1, the number is necessarily negative: the other seven bits can add at most 127 and cannot get past −128.

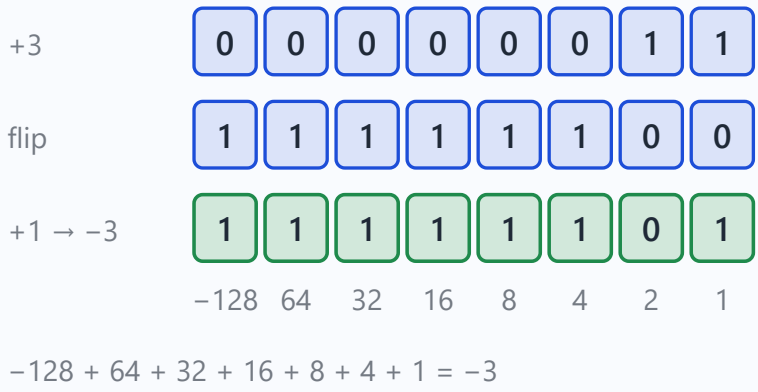


Figure 1.1.8: Reading −3 using weights. Taking the weight of the leftmost bit as −128 gives $-128 + 64 + 32 + 16 + 8 + 4 + 1 = -3$.

Example 1.1.6

What is the bit pattern `10110110` as an unsigned and as a signed number?

Read as unsigned, all weights are positive: *128 + 32 + 16 + 4 + 2 = 182*.

Read as signed, the weight of the leftmost bit is −128: *−128 + 32 + 16 + 4 + 2 = −74*. The same result follows from *182 − 256 = −74*. When the leftmost bit is 1, the signed value is always $2^8 = 256$ less than the unsigned value, because that bit contributes −128 instead of +128, a difference of 256.

This observation gives a general rule. In an n-bit two's complement number, $-x$ is stored as $2^n - x$ when read as unsigned. That is why −20 appears as 236 and −3 as 253. It helps to picture the numbers arranged like a clock face: the positives run clockwise from 0 up to 127, then come −128 and the negatives up to −1. One step after −1 you are back at 0.

1.1.3.3 Two's complement in Java

All of Java's integer types (`byte` , `short` , `int` , `long`) use two's complement. This is stated explicitly in the Java language specification and is the same on every platform. The following program tests the same rule in Java.

```
int n = 3;
int neg = ~n + 1; // invert, add 1
System.out.println(neg);
byte b = (byte) 0b11111101;
System.out.println(b);
System.out.println(Integer.toBinaryString(-3));
System.out.println(Byte.MIN_VALUE + " .. "
    + Byte.MAX_VALUE);
```

```
-3
-3
11111111111111111111111111111101
-128 .. 127
```

The `~` operator flips all bits of an integer; it is the **bitwise NOT** operator. `~n + 1` is exactly the two's complement rule, and the result is −3. In Java, `-x` and `~x + 1` give the same value for every integer.

In the second part, the literal `0b11111101` is converted to a `byte` . To Java this literal is an `int` with the value 253. Since 253 does not fit in the range of `byte` , an explicit **cast** is needed: `(byte)` . The cast keeps only the rightmost 8 bits; these are `11111101` , which read as signed is −3.

`Integer.toBinaryString(-3)` prints all thirty-two bits: thirty 1s at the start and `01` at the end. The representation does not change when it is widened to 32 bits; the bits added on the left are copies of the sign bit. This rule is called **sign extension**. A positive number is extended to the left with zeros

and a negative number with ones, and the value is preserved. The last line prints the limits of the `byte` type: `-128 .. 127`.

That Java's `byte` is signed can sometimes be surprising. Raw bytes read from the network or a file are often thought of as lying between 0 and 255. In Java, however, a byte with the value 200 appears as −56. To read a byte as unsigned, use the expression `b & 0xFF`: the `&` operation clears the ones that sign extension brought in and gives back 200. The `Byte.toUnsignedInt(b)` method does the same job.

1.1.3.4 Overflow

> **Definition 1.1.4**
>
> **Overflow** is the situation in which the result of an operation does not fit in the range of the type. In Java, integer overflow raises no error: the result wraps around "like a clock" and continues from the other end of the range.

```
byte b = 127;
b++;
System.out.println(b);
int m = Integer.MAX_VALUE;
System.out.println(m + 1);
System.out.println(Math.addExact(m, 1));
```

```
-128
-2147483648
Exception in thread "main" java.lang.ArithmeticException: integer overflow
    at java.base/java.lang.Math.addExact(Math.java:911)
    at Overflow.main(Overflow.java:7)
```

In the first three lines, `b` is 127, and after being incremented by 1 it should become 128. Adding 1 to 127 = `01111111` gives `10000000`. Read as signed, this bit pattern is −128. The adder worked correctly; only the result moved into the negative region because of the carry into the sign bit. On the clock face, one step forward from 127 lands on −128. The same happens with `Integer.MAX_VALUE + 1`, which gives -2^{31} = −2,147,483,648.

The danger of overflow is that it is silent. The program does not crash or warn; it simply keeps running with a wrong number. If a bank balance or a counter overflows, the error shows up much later, somewhere else. That is why Java offers the `Math.addExact`, `Math.subtractExact` and `Math.multiplyExact` methods for checked operations. When these methods see that the result does not fit, they throw an `ArithmeticException` and the program stops at that point. The call on the last line therefore prints no number; the exception message reports the place and cause of the error. The

`at` lines in the output show which method calls the exception was thrown through. Here the code was placed in the `main` method of a class named `Overflow` and run, and the line numbers refer to that file.

One way to prevent overflow is to use a wider type. The product of two `int`s fits in a `long`, but the multiplication itself must be done as a `long`. Writing `long x = 100000 * 100000;` does not help: the right-hand side is a product of two `int`s, computed first in 32 bits, where it overflows; only then is the overflowed result assigned to a `long`. The right way is to make one of the factors a `long`: `100000L * 100000`. The `L` suffix on the literal makes it a `long`, and the whole product is carried out in 64 bits.

Example 1.1.7

What is `01101101 + 00110110`, and how is the result read as an 8-bit signed `byte`?

In decimal, *109 + 54 = 163*. Adding the bits gives `10100011`: *128 + 32 + 2 + 1 = 163*. No carry spilled into a ninth bit; read as unsigned, the result is correct. Read as signed, however, the leftmost bit is 1 and the value is *163 − 256 = −93*. The sum of two positive numbers came out negative. This is exactly the sign of overflow in signed addition: two numbers of the same sign are added and the result has a different sign. In Java, the expression `(byte) (0b01101101 + 0b00110110)` gives −93.

1.1.4 Fractional numbers: floating point

1.1.4.1 Fraction and exponent

With a limited number of digits, only a limited number of values can be represented. A 32-bit `int` counts up to about ±2 billion. That is not even enough to square the speed of light (about $3{\cdot}10^{8}$ m/s). And it cannot write the charge of an electron (about $1.6{\cdot}10^{-19}$ coulomb) at all. Engineering calculations need both very large and very small numbers.

The solution is the idea behind scientific notation. A number is split into two parts: the **mantissa** (also called the significand), which carries the significant digits, and the **exponent**, which fixes the position of the decimal point. The parts are stored separately. Suppose `31415901` is written in a field of eight digits. Let the first six digits (314159) stand for the mantissa and the last two digits (01) for the power of 10, that is, 10^{1}. If the mantissa is read as 0.314159, the number is $0.314159 \times 10^{1} = 3.14159$.

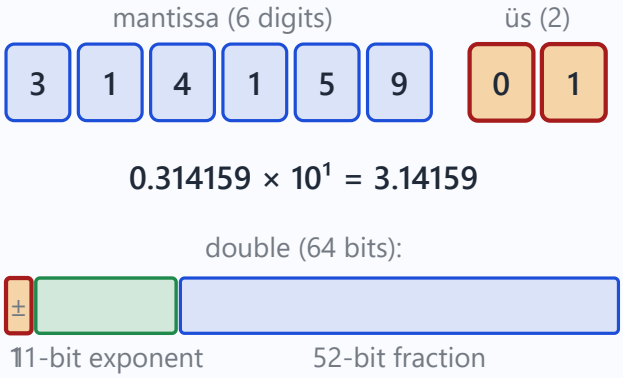


Figure 1.1.9: At the top, six of the eight digits are used as the mantissa and two as the exponent. At the bottom, Java's `double` type: 1 sign bit, 11 exponent bits and 52 fraction bits.

The name of this representation comes from here: **floating point**. As the exponent changes, the decimal point floats. If the same mantissa is multiplied by 10^3, we get $0.314159 \times 10^3 = 314.159$. The digits are the same; only the position of the point has moved. In this way, a limited number of digits can write both very large and very small numbers. The price is that every number is stored with only as many significant digits as fit in the mantissa, that is, *approximately*.

1.1.4.2 IEEE 754 and Java's float and double

Computers apply the same idea in binary. The rules are set by the **IEEE 754** standard, which almost all processors follow today. Java has two floating-point types: `float` is 32 bits and `double` is 64 bits.

Type	Sign	Exponent	Fraction	Significant decimal digits	Largest value
`float`	1	8	23	about 7	$\approx 3{,}4 \cdot 10^{38}$
`double`	1	11	52	about 15–16	$\approx 1{,}8 \cdot 10^{308}$

Table 1.1.7: Bit layout of Java's floating-point types.

In binary floating point, a number is put into the form $(-1)^{sign} \times 1.fraction_2 \times 2^{exponent}$. Since every nonzero binary number can be shifted so that it starts with "1....", the leading 1 is not stored; the fraction field holds only the bits after the point. The exponent is written as a non-negative number by adding a constant, called the bias; in `float` this constant is 127, and in `double` it is 1023.

Example 1.1.8

The `float` representation of −6.5 is found in five steps.

1. The sign is negative: the sign bit is 1.
2. 6.5 = 4 + 2 + 0.5. In binary it is written 110.1_2 (the weight of the first place after the point is $2^{-1} = 0.5$).
3. The point is shifted two places to the left: $110.1_2 = 1.101_2 \times 2^2$.

4. The exponent is 2; adding 127 gives 129 = `10000001`, which is what is stored.
5. The fraction is `101` followed by 20 zeros, with the leading 1 dropped.

The result is `1 10000001 10100000000000000000000`. In Java, `Integer.toBinaryString(Float.floatToIntBits(-6.5f))` prints exactly these 32 bits.

Numbers like −6.5 are written exactly in binary, because 0.5 is a power of two. 0.1 is not. Just as 1/3 goes on forever in decimal as 0.3333..., 0.1 repeats forever in binary as $0.000110011001100\ldots_2$. Because the fraction field is finite, it is cut off somewhere and rounded. The 0.1 in memory is really a number very close to 0.1, but not equal to it.

1.1.4.3 Rounding error and comparison

The best-known consequence of rounding is that in Java `0.1 + 0.2` comes out as `0.30000000000000004`. The small errors in the stored values of 0.1 and 0.2 add up, and the result is not equal to the stored value of 0.3. So the expression `0.1 + 0.2 == 0.3` gives `false`. This is not a Java bug; the same result appears in C, Python and JavaScript, because they all use the same IEEE 754 hardware.

For this reason floating-point numbers are not compared with `==`. Instead, we check whether the difference of the two numbers is small enough:

```
double a = 0.1 + 0.2;
double b = 0.3;
System.out.println(a == b);                    // false
System.out.println(Math.abs(a - b) < 1e-9);    // true
```

The bound 10^{-9} here is called the **tolerance**, and it is chosen according to the scale of the problem. For a voltage measured in volts, 10^{-9} is a very small difference; but for a capacitance on the order of 10^{-12} farad it is very large. For values such as money, which must be correct to the last kuruş, `double` is never used. Either amounts are stored as a `long` number of kuruş, or Java's `BigDecimal` class, which computes in decimal, is used.

Common mistake

Tying a loop to an equality condition on a floating-point counter: `for (double x = 0; x != 1.0; x += 0.1)`. It looks as if `x` will be 1.0 after ten steps, but because of accumulated rounding errors `x` becomes 0.9999999999999999 and is never equal to 1.0; the loop runs forever. Counters should be integers, or the condition should be written as an inequality such as `x < 1.0`.

1.1.5 Letters and strings

1.1.5.1 Character encoding and ASCII

A computer works only with numbers. For it to process text, each letter must be assigned a number. Which letter is shown by which number is an arbitrary choice, but everyone has to make the same choice. Otherwise the number written as "A" on one computer would be read as a different letter on another.

> **Definition 1.1.5**
>
> A **character encoding** is the rule that determines which numbers characters are stored as, and with which byte sequences in memory.

The oldest and most influential encoding is **ASCII** (American Standard Code for Information Interchange). ASCII was published in 1963 by the American standards body ASA (today's ANSI). It represents uppercase and lowercase English letters, digits, punctuation marks and some special characters with numbers.

ASCII is a 7-bit code and defines $2^7 = 128$ characters (0–127). Codes 0–31 and 127 are **control characters**, which do not appear on screen. They were originally designed to control printers and teletype machines: return to the start of the line, move down one line, ring the bell. Codes 32–126 are the printable characters. Table 1.1.8 shows a few important codes.

Character	Decimal	Binary
`'A'`	65	`1000001`
`'a'`	97	`1100001`
`'0'`	48	`0110000`
space	32	`0100000`
newline `\n`	10	`0001010`

Table 1.1.8: Some ASCII codes.

The layout in the table was chosen deliberately. Letters are arranged in consecutive codes in alphabetical order: `'A'` = 65, `'B'` = 66, …, `'Z'` = 90. This turns comparing letters into comparing numbers. A lowercase letter is 32 more than its uppercase letter: `'a'` = 97 = 65 + 32. In binary this difference is a single bit, the 2^5 place. `'A'` = `1000001` and `'a'` = `1100001`. Converting between uppercase and lowercase is just switching one bit on or off. Digits are also arranged starting from 48; the numeric value of a digit character is found with `c - '0'`: `'7' - '0'` = 55 − 48 = 7.

In Java, `char` is a numeric type and can take part in arithmetic. The expression `(char) ('a' - 32)` gives `'A'`. This closeness can be surprising at times: the line `System.out.println('a' + 'b');` prints `195`, not `ab`. The sum of two `char`s is an `int`: *97 + 98 = 195*. If the same line is written as `"" + 'a' + 'b'`, `'a'` is first added to the empty string, and since the result is a string, `'b'` is appended too, giving `ab`. In Java the `+` operator works from left to right and turns into concatenation as soon as one of the operands is a string.

Seven bits leave one of the eight bits of a byte unused. Eight-bit extensions use this bit to place extra letters in 128–255, reaching 256 symbols. ISO 8859-9 (Latin-5) and Windows-1254 are encodings of this kind for Turkish; they place the letters ğ, ş, ı and İ in this upper half. But trouble arose once every language defined its own upper half. The same value 253 was the letter ı in one table and ý in another. When a Turkish text was opened with the wrong encoding, the letters looked garbled. A single 8-bit table cannot serve all the world's languages; Chinese and Japanese have thousands of characters.

1.1.5.2 The ASCII table

When the printable ASCII characters are laid out in a table of 16 columns, their codes are easy to read (Table 1.1.9). The code of a character is the sum of its row header and its column header. For example, `'J'` is in row 64 and column 10: *64 + 10 = 74*.

+	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
32	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Table 1.1.9: Printable ASCII characters (32–126). Code = row + column; SP is the space character.

The last cell of the table, DEL (127), is a control character. Three of the control characters between 0 and 31 are still used every day: 9 tab (`\t`), 10 newline (`\n`) and 13 carriage return (`\r`). In Windows text files the end of a line is marked with the pair `\r\n`, and in Linux and macOS with `\n` alone. This difference is often why a file looks right on one system and like a single line on another. The complete 0–255 range, and a search by character or code, are on the ASCII Table page.

1.1.5.3 Unicode

The confusion created by country-specific code tables gave rise to the idea of a single table for all languages.

Definition 1.1.6

Unicode is the standard that gives a number to every letter, symbol and mark of all the world's writing systems. This number is called a **code point** and is written in hexadecimal with the prefix `U+` : U+0041 = A, U+00E7 = ç.

Unicode has been developed by the Unicode Consortium since 1991. Its international counterpart, the ISO/IEC 10646 standard, is kept in step with it. The first version was 16 bits, and it was thought that 65,536 characters would be enough. When historical scripts, mathematical symbols and all the Chinese-Japanese-Korean ideographs were added, that space proved too small. The code space was extended to U+10FFFF, giving 1,114,112 code points; today more than 150,000 of them have been assigned characters.

The first 128 code points of Unicode are identical to ASCII: U+0041 is 65 in both ASCII and Unicode. The first 256 code points also coincide with Latin-1 (ISO 8859-1), the table used for Western European languages; so ç is 231 in both Latin-1 and Unicode. The Turkish-specific ğ, ş and ı are not in Latin-1 and lie above 256 (Table 1.1.10).

Character	Code point	Decimal	UTF-8 bytes
A	U+0041	65	`41`
ç	U+00E7	231	`C3 A7`
ğ	U+011F	287	`C4 9F`
€	U+20AC	8364	`E2 82 AC`
😀	U+1F600	128512	`F0 9F 98 80`

Table 1.1.10: Code points and UTF-8 bytes of some characters (hexadecimal).

A code point is a number; it does not say how to store it in memory. The simplest way is to store every character in a fixed 32 bits. This encoding is called UTF-32. But in an English text every letter has a 7-bit ASCII code, and reserving 32 bits wastes three quarters of the space. This is why variable-length **UTF** (Unicode Transformation Format) encodings were defined.

1.1.5.4 UTF-8

UTF-8 stores a code point in a length that varies from 1 to 4 bytes. Frequently used characters take few bytes and rare ones take many. ASCII characters are a single byte, and that byte is the same as the ASCII code. So a pure ASCII text is already a valid UTF-8 text with no change; UTF-8 is **backward compatible** with ASCII. Today the great majority of web pages are encoded in UTF-8.

A UTF-8 reader can tell how many bytes a character will take from the bits at the start of the first byte. Table 1.1.11 gives the patterns. The bits of the code point are placed in order into the positions marked `x` .

Code point range	Byte pattern	Number of bytes	Bits carried
U+0000–U+007F	`0xxxxxxx`	1	7
U+0080–U+07FF	`110xxxxx 10xxxxxx`	2	11
U+0800–U+FFFF	`1110xxxx 10xxxxxx 10xxxxxx`	3	16
U+10000–U+10FFFF	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`	4	21

Table 1.1.11: UTF-8 byte patterns.

The design of the patterns is clever. A byte starting with 0 is an ASCII character on its own. A byte starting with 110, 1110 or 11110 is the first byte of a character, and the number of leading 1s gives the total number of bytes. A byte starting with 10 is always a continuation byte. Even if a reader starts reading in the middle of a text, it can skip the continuation bytes and find the start of the next character.

Example 1.1.9

The UTF-8 encoding of the character ç is found as follows. The code point is U+00E7 = 231, and since it lies in the range U+0080–U+07FF it uses two bytes. 231 is written in 11 bits as `00011 100111`. The first five bits go into the `110xxxxx` pattern of the first byte, and the last six bits into the `10xxxxxx` pattern of the second byte:

> `110 00011` = C3 `10 100111` = A7

The result is C3 A7. In the same way, € (U+20AC = `0010 000010 101100`) fits into three bytes: `1110 0010` = E2, `10 000010` = 82, `10 101100` = AC.



Figure 1.1.10: The UTF-8 encoding of the character ç. The colored boxes are the fixed bits of the pattern and the others are bits of the code point; the two bytes are hexadecimal C3 and A7.

How many bytes a string takes in UTF-8 can be seen with the `getBytes` method. For the code to compile, the line `import java.nio.charset.StandardCharsets;` must be added at the top of the file.

```
String word = "çikolata";
System.out.println(word.length());
System.out.println(word.getBytes(
    StandardCharsets.UTF_8).length);
```

```
8
9
```

`"çikolata"` ("chocolate") has eight characters, and `length()` gives 8. In UTF-8, ç takes two bytes and each of the other seven letters one byte: *2 + 7 = 9*. This is why the size of a file in bytes is not the same as the number of characters in it.

Encoding determines not only length but also readability. A text must be read with the same encoding it was written in. "ç" written in UTF-8 is two bytes (C3 A7). If these two bytes are read with the Windows-1254 table, each is taken as a separate character and "Ã§" appears on screen. This is the problem in most files where Turkish characters look garbled. When reading and writing files in Java, always stating the encoding explicitly, for example by passing `StandardCharsets.UTF_8`, prevents this error.

1.1.5.5 char, UTF-16 and emoji in Java

When Java was designed, Unicode was still 16 bits. Java's `char` type was therefore chosen to be 16 bits, on the assumption that every character would fit in a single `char`. When Unicode grew, this assumption broke. Today a `char` is not a character but a **UTF-16 code unit**. The UTF-16 encoding writes characters up to U+FFFF as a single 16-bit unit; ş = 01 5F, € = 20 AC. Larger code points are written with two units. This pair is called a **surrogate pair**.

Most everyday text does not feel this distinction, because the common characters of Turkish, Russian, Arabic and Chinese all lie below U+FFFF. The difference shows up most with emoji. Emoji are symbols that quickly convey feelings, expressions and objects, and computers handle them as text, not as images. Their standard is set by the Unicode Consortium, and each emoji is a unique code point: 😀 = U+1F600. The code is the same on every device, so a sent emoji is read with the same meaning on the other end. Its drawing, the **glyph**, may look different depending on the font and the vendor.

Some emoji are combinations of several code points. The family emoji 👨‍👩‍👧 consists of five code points: man, ZWJ, woman, ZWJ, girl. ZWJ (zero width joiner) is U+200D; it is invisible on screen and only asks that the neighboring emoji be drawn as a single glyph. A system that does not recognize this joiner shows three separate faces.

```
String face = "😀";
System.out.println(face.length());
System.out.println(
    face.codePointCount(0, face.length()));
String family = "👩‍👩‍👧";
System.out.println(family.length());
```

```
2
1
8
```

The string 😀 has a `length()` of 2, because U+1F600 is a surrogate pair of two `char`s: `\uD83D` and `\uDE00`. `codePointCount`, on the other hand, counts code points and gives 1. In the family emoji, the three person emoji take two `char`s each and the two ZWJs one `char` each: *3·2 + 2·1 = 8*. Although a single symbol appears on screen, `length()` is 8 and the number of code points is 5. In UTF-8 this emoji takes *3·4 + 2·3 = 18* bytes.

The conclusion of this section for Java is this: `String` stores characters in UTF-16, and `length()` is the number of `char`s, not the "number of characters". In Turkish text the two are almost always the same. In text with emoji, mathematical symbols or historical scripts they differ. To iterate over the characters of such text one by one, use `codePoints()` or `codePointAt` instead of `charAt`. Since Java 9, `String` stores strings containing only Latin-1 characters internally with one byte per character. This is a memory optimization; `length()` and `charAt()` continue to work in UTF-16 units in every case.

Common mistake

Taking the `length()` of a string to be the number of symbols shown on screen or the number of bytes. `"😀".length()` is 2, not 1; `"ağaç".length()` is 4, while the same string takes 6 bytes in UTF-8. If you need to check whether it fits in a database field or a network packet, look at the number of encoded bytes; if you need to compute a length to show the user, look at code points.

1.1.5.6 Code trace: the character codes of a string

A string sits in memory as two parts: an array in which the characters are laid out as numbers, and the length of that array. The length is kept separately; this is how it is known where the text ends. The following program prints the code of each character of the string `"çikolata"`.

```
String word = "çikolata";
for (int i = 0; i < word.length(); i++) {
  char ch = word.charAt(i);
  System.out.print((int) ch + " ");
}
System.out.println();
System.out.println("Length: " + word.length());
```

The loop starts at `i = 0` and runs as long as `i < word.length()` is true. `charAt(i)` returns the `char` at position `i`. The cast `(int) ch` prints the numeric code of the character rather than the character itself; without the cast, the letter itself would be printed. Unlike `println`, `print` does not end the line, so the codes are lined up side by side. The `println()` after the loop ends the line. Table 1.1.12 shows each iteration.

`i`	**Condition** `i < 8`	`ch`	`(int) ch`	**Explanation**
0	true	`'ç'`	231	U+00E7; not in ASCII
1	true	`'i'`	105	
2	true	`'k'`	107	
3	true	`'o'`	111	
4	true	`'l'`	108	
5	true	`'a'`	97	
6	true	`'t'`	116	
7	true	`'a'`	97	same code as the `'a'` at position 5
8	false	The loop ends; a line break and `Length: 8` are printed.		

Table 1.1.12: Trace of the loop for `"çikolata"`.

```
231 105 107 111 108 97 116 97
Length: 8
```

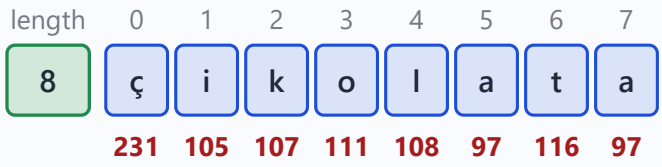


Figure 1.1.11: The string `"çikolata"` in memory: on the left the length (8), on the right the characters with indices 0–7 and their codes below them.

All the codes other than ç are values from the ASCII table. 231 is the Unicode code point of ç (U+00E7). The same value is 231 in the Latin-1 and Turkish ISO 8859-9 tables too, so the program prints the same number whichever table one has in mind. The expression `(int) ch` gives the code point, not the UTF-8 bytes. The UTF-8 form of ç is the bytes C3 A7, but in Java memory ç sits as a single 16-bit `char` with the value 231.

1.1.6 Structured information

1.1.6.1 Images, sound and video

Just as numbers represent letters and letters represent text, sequences of numbers can represent far more complex information. Whatever a file stores, to the computer it is a sequence of bytes. The meaning of the sequence comes from the rules that the programs writing and reading it follow, that is, from the **file format**. Most of our interaction with computers rests on these rules, which let us see sequences of numbers as images, sound or video.

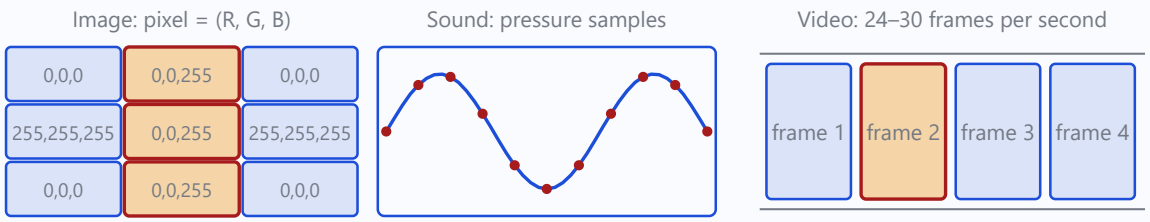


Figure 1.1.12: Three examples of structured information. On the left, an image whose every pixel is an (R, G, B) triple; in the middle, sound pressure measured at regular intervals over time; on the right, video made of successive images at 24–30 frames per second.

- **Image.** An image consists of tiny dots arranged in rows and columns, called **pixels**. Each pixel is three numbers giving the amounts of red, green and blue light, and each number lies between 0 and 255. This triple is called RGB (red, green, blue). A blue pixel is (0, 0, 255), a white pixel is (255, 255, 255) and a black pixel is (0, 0, 0). Each color component fits in one byte, and a pixel takes three bytes. The color code `#FF0000` used on the web is the hexadecimal notation of these three bytes: FF red, 00 green, 00 blue.
- **Sound.** Sound is the change of air pressure over time. A microphone turns this change into a voltage. An analog-to-digital converter (ADC) measures the voltage at regular intervals and records each measurement as a number. This process is called **sampling**. CD quality takes 44,100 samples per second, and each sample is a 16-bit number. The choice of sampling rate is not arbitrary. The human ear hears up to about 20 kHz. To sample a signal without losing information, one must also measure faster than twice its highest frequency.

- **Video.** A video is a sequence of images shown at 24 or 30 frames per second. At this rate the eye perceives the images as continuous motion. Each frame is a sequence of numbers like an image; the video is a sequence of those sequences, with a sound channel added alongside.

1.1.6.2 How much space does a video take?

Computing the sizes of these numbers makes the cost of the representation concrete. The following program computes how many bytes one minute of uncompressed Full HD video and one minute of CD audio take.

```
long pixels = 1920L * 1080;  // one Full HD frame
long frame = pixels * 3;      // R, G, B: 3 bytes
long perSecond = frame * 30;     // 30 frames/s
System.out.println("Pixels: " + pixels);
System.out.println("Frame (bytes): " + frame);
System.out.println("1 second (bytes): " + perSecond);
System.out.println("1 minute (MB): "
    + perSecond * 60 / 1_000_000);
// 44.1 kHz, 16 bits (2 bytes), stereo, 1 min
long audio = 44_100L * 2 * 2 * 60;
System.out.println("1 min CD audio (bytes): " + audio);
```

```
Pixels: 2073600
Frame (bytes): 6220800
1 second (bytes): 186624000
1 minute (MB): 11197
1 min CD audio (bytes): 10584000
```

The computation goes step by step. A Full HD frame is *1920 × 1080 = 2,073,600* pixels. Since each pixel takes three bytes, a frame is *2,073,600 × 3 = 6,220,800* bytes, about 6.2 MB. Thirty frames per second come to *6,220,800 × 30 = 186,624,000* bytes. A minute is 60 times that: 11,197,440,000 bytes. The program divides this by `1_000_000` to get megabytes. Because integer division discards the fraction, the output is 11197 MB; that is, one minute of uncompressed Full HD video takes about **11 GB**. On the audio side, multiplying 44,100 samples, 2 bytes and 2 channels (stereo) gives 176,400 bytes per second and 10,584,000 bytes per minute, about 10.6 MB.

In real life, video and audio are stored **compressed**. Video codecs such as H.264 exploit the similarity between successive frames, and audio codecs such as MP3 exploit details the ear cannot hear, shrinking the file many times over. But a compressed file is still a sequence of numbers; only the rules that give the numbers their meaning change.

The most important detail of the code from Java's point of view is the `L` suffix in the first line. Writing `1920L * 1080` makes the multiplication a `long` multiplication. Without the `L` suffix, all the products would be computed as `int`. The one-second value (186,624,000) fits in an `int`, but the

one-minute value (11,197,440,000) exceeds `Integer.MAX_VALUE` by more than a factor of five. The expression `1920 * 1080 * 3 * 30 * 60` therefore overflows silently and gives a meaningless negative number such as −1,687,461,888. The `44_100L` in the audio computation is written with the `L` suffix for the same reason; there the result would fit in an `int`, but writing the suffix is a safe habit.

1.1.7 The stored program

1.1.7.1 The stored-program concept

So far we have seen how numbers represent data. The last step is that the instructions telling the processor what to do are numbers too. This idea is the basic principle of modern computers.

> **Definition 1.1.7**
>
> The **stored-program concept**: instructions are numbers like data, and they are stored in the same memory as the data. The instructions and data held in memory together make up the program. A machine that works on this principle is called a **stored-program computer**.

Each instruction represents a specific task or operation: read from memory, add, write to memory. Instructions are normally processed one after another, in the order they sit in memory. But when a condition holds, the program can **jump** to a different instruction. This is what lets programs follow different paths depending on the situation. The `if` statement and loops in Java turn into exactly this kind of jump instruction at the processor level.

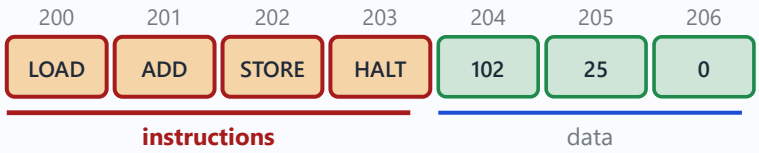


Figure 1.1.13: Instructions and data sit in the same memory, at neighboring addresses. The LOAD, ADD, STORE and HALT instructions are at addresses 200–203, and the data at addresses 204–206.

The importance of the idea becomes clear when compared with the machines before it. ENIAC, completed in 1945, was one of the first large electronic computers, able to perform thousands of additions per second. But its program lived not in memory but in wires and switches. For a new computation, operators had to replug the cables on the panels by hand, and this could take days.

The EDVAC design, which the ENIAC team was working on in the same year, proposed a different path: instructions and data would be stored in memory in sequence. Changing the program meant writing new numbers into memory. This approach was flexible and extensible; a program could even

read and produce another program as data. Compilers do exactly this: they read source code and produce machine code. The idea was spread by John von Neumann's 1945 report "First Draft of a Report on the EDVAC". The first machine to run a stored program was a small experimental computer at the University of Manchester in 1948, known as the "Baby".

1.1.7.2 The memory hierarchy

Memory is where running programs and their data are temporarily kept. An ideal memory would be very large, very fast and cheap at once. Unfortunately these three properties conflict. Fast memory technologies are expensive and take up a lot of room on the chip. Cheap, large memories are slow. Computers resolve this conflict by arranging memories of different speeds and sizes in layers.

Definition 1.1.8

The **memory hierarchy** is the arrangement of memories from levels that are close to the processor, fast and small to levels that are far, slow and large. The order is: registers, the L1, L2 and L3 caches, main memory (RAM) and disk.

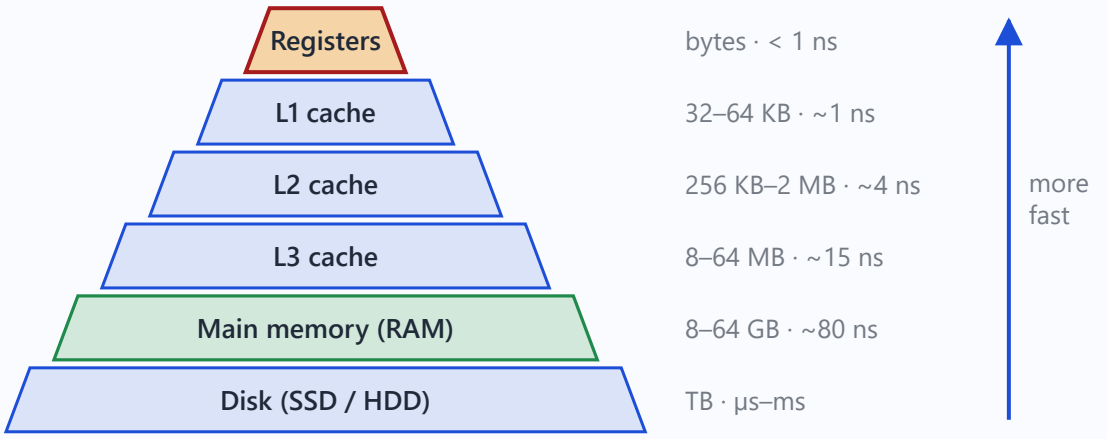


Figure 1.1.14: The memory hierarchy. Moving up, memory gets smaller and faster. Sizes and access times are approximate and vary with the processor; 1 ns is one billionth of a second.

- **L1 cache**: the fastest cache, located inside the processor. Its size is a few tens of kilobytes per core, and it is accessed in about 1 ns.
- **L2 cache**: backs up L1 and is larger than it. Most processors also have an **L3** cache of a few tens of megabytes, shared by all the cores.
- **Main memory (RAM)**: gigabytes in size, but tens of times slower to access than the caches.
- **Disk**: terabytes in size and permanent; the access time is between microseconds and milliseconds.

The hierarchy works because programs do not access memory randomly but with a certain pattern. A program tends to reuse data it used a short while ago; loop variables are an example. A program that accesses one address often accesses the addresses right next to it too; walking through an array from start to end is an example. These two tendencies are called **locality**. A cache keeps recently used data and its neighbors next to the processor. In this way the great majority of accesses are served from the fast cache without going to the slow RAM. Memory close to the processor is faster but smaller and more expensive; how this hierarchy is used directly affects a program's performance. If one of two Java programs that perform the same number of additions walks through an array in order and the other jumps around at random, a speed difference of several times can be seen between them.

1.1.7.3 The processor and the bus

The **central processing unit** (CPU) is where all of the computer's computation is done. It has three basic parts.

- **Arithmetic logic unit** (ALU): does computations such as addition, subtraction, comparison and bitwise logic operations. The full adders described under binary addition are inside the ALU.
- **Control unit**: interprets instructions, sequences the operations and directs them. It decides which register will be connected to the ALU and whether a read or a write signal goes to memory.
- **Registers**: very small but very fast storage areas for temporary data. The numbers the ALU works on come from registers, and the results are written to registers.

The structure that connects the processor to memory and to input/output devices is called the **bus**. The bus is a communication system made of many parallel wires. The wires fall into three groups according to their function.

- **Address signals** (address bus): select which memory cell or device is accessed.
- **Data signals** (data bus): carry the actual data to be processed or stored.
- **Control signals** (control bus): direct the operation; for example, they signal whether the operation is a "read" or a "write".

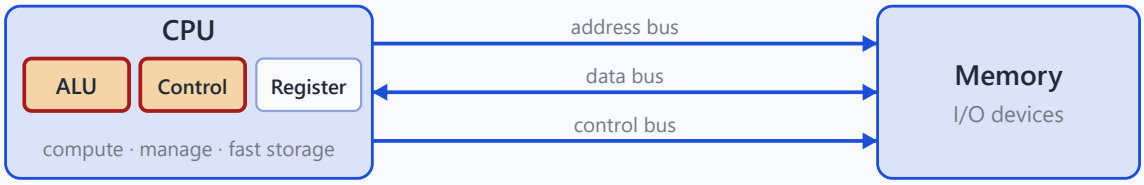


Figure 1.1.15: The address, data and control buses between the processor (ALU, control unit, registers) and memory and I/O devices.

The number of wires in the address bus determines how much memory can be addressed. Each wire carries one bit; an address bus with n wires can select 2^n different addresses. A 32-line address bus selects $2^{32} = 4{,}294{,}967{,}296$ different addresses. If each address points to one byte, that amounts to 4

GiB of memory. This is why 32-bit systems can use at most about 4 GB of RAM, and this limit is one of the main reasons for the move to 64-bit processors. Seen through Theorem 1.1.1, each wire of an address bus is like a one-bit digit, and n digits can write 2^n different numbers.

1.1.7.4 Registers

Figure 1.1.16 summarizes how the processor, memory and bus work together. Every memory cell has an address and a content. The registers inside the processor are the layer of memory closest to the processor, the smallest and the fastest.

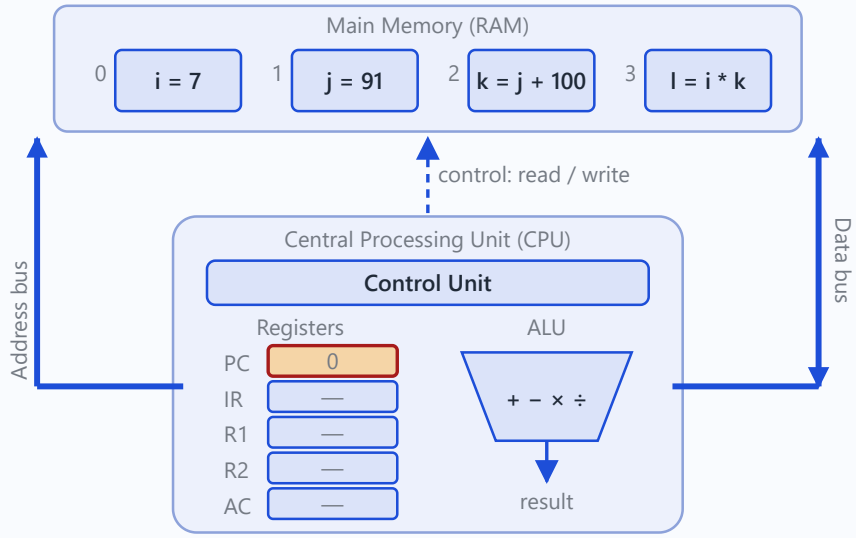


Figure 1.1.16: At the top, main memory (four cells at addresses 0–3); at the bottom, the processor: control unit, the PC, IR, R1, R2 and AC registers, and the ALU. On the left is the address bus, on the right the bidirectional data bus, and in the middle the control signal.

- The processor tells the address bus which cell it wants.
- The contents of the cell travel over the data bus to the processor (a read) or from the processor to the cell (a write).
- **PC** (program counter): holds the address of the next instruction.
- **IR** (instruction register): holds the instruction currently being decoded.
- **R1, R2** and **AC** (accumulator): hold intermediate results.
- The ALU computes with the numbers in the registers.

In electronics, registers are built from flip-flops. A flip-flop stores one bit; 32 flip-flops side by side make a 32-bit register. Registers can take a new value on every pulse of the clock signal, so they can be read and written within one clock cycle. There are few of them: a typical processor has a few dozen general-purpose registers. One of the compiler's important jobs is to keep frequently used variables in registers rather than in memory.

1.1.7.5 Memory, addresses and the program counter

Memory is the area where data and instructions are stored. From the processor's point of view, a program is a long list of instructions laid out one after another in memory. Every memory cell has an **address**, and the cell holds an instruction or a piece of data encoded as a number. Instructions are fetched from memory one at a time, loaded into the processor's registers and executed in order.

> **Definition 1.1.9**
>
> The **program counter** (PC) is the register that holds the address of the next instruction. After the processor fetches an instruction, it advances the PC to the next address.

The processor normally proceeds in address order: 200, 201, 202 … But some instructions change this order. In the small piece of memory in Table 1.1.13, there is a `JUMP 7` instruction at address 204. In this example, 7 says how far ahead of the instruction's own address to jump: *204 + 7 = 211*. The instruction writes 211 into the PC, and the processor skips 205–210 and continues with the instruction at 211. On some machines the jump target is given as such a relative offset, on others directly as an absolute address (like `JUMP 211`); in both cases the result is that the PC changes.

Address	Content	Kind
200	`ADD to R1`	instruction
201	`102`	data
202	…	
203	…	
204	`JUMP 7`	instruction (jump)
…	…	
211	next instruction	instruction

Table 1.1.13: Instructions and data side by side in memory.

The "Kind" column of the table is only there to help the reader; memory has no such column. Whether a memory cell holds an instruction or data cannot be told by looking at the cell: both are numbers. The difference is how the processor uses the cell. When the PC points to a cell, its content is decoded as an instruction; when an instruction reads the cell, its content is used as data. If the PC accidentally points to a data cell, the processor takes that number for an instruction and does meaningless things. This is why computing jump targets correctly is so important.

1.1.7.6 The fetch–decode–execute cycle

Definition 1.1.10

The **fetch–decode–execute cycle** is the basic operating cycle that the processor repeats for every instruction. It describes how programs are executed on the processor.

1. **Fetch**: the processor takes the instruction at the address the PC points to from memory and writes it into the IR. The PC advances to the next address.
2. **Decode**: the instruction is decoded. Which operation is to be done, which registers are to be used, which address is to be accessed? Every instruction has a specific function, and the control unit reads this function from the instruction's bits.
3. **Execute**: the operation the instruction calls for is carried out. This may be processing data, computing a result, reading from memory or writing to memory.

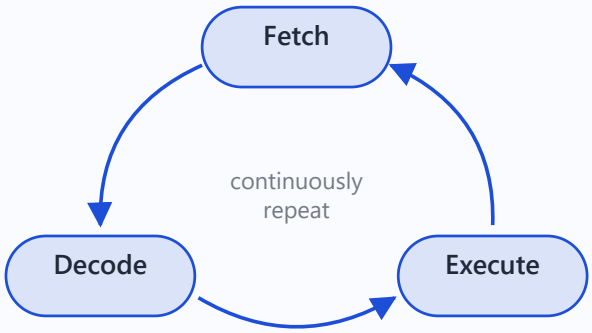


Figure 1.1.17: The cycle made of the Fetch, Decode and Execute steps. It repeats continuously as long as the computer is on.

Then the next instruction is fetched and the cycle begins again. This cycle repeats without stopping as long as the computer is on. The whole run of a Java program is nothing but this cycle turning billions of times.

The cycle is driven by a **clock** signal. The clock of a 3 GHz processor "ticks" 3 billion times a second; one clock cycle lasts about 0.33 ns. In a simple processor, each instruction takes a few clock cycles. Modern processors, however, overlap the processing of several instructions: while one instruction is executed, the next is decoded and the one after is fetched. This arrangement is called a **pipeline** and resembles an assembly line in a factory. When the line is full, one instruction completes in every clock cycle.

1.1.7.7 Code trace: 102 + 25 on a toy computer

The operation of the cycle is easy to follow on a toy computer. In this computer, memory addresses 200–203 hold four instructions and addresses 205–207 hold data. The program adds the number at 205 to the number at 206 and writes the result to 207. The parts after the semicolon on each line are comments, and the notation `[x]` means "the memory cell at address x".

```
200: LOAD  R1, [205] ; R1 ← mem[205]
201: ADD   R1, [206] ; R1 ← R1 + mem[206]
202: STORE R1, [207] ; mem[207] ← R1
203: HALT            ; stop
204: ...
205: 102             ; data
206: 25              ; data
207: 0               ; result here
```

The instructions mean the following. `LOAD` copies a value in memory into a register. `ADD` adds a value in memory to the value in a register. `STORE` writes the value in a register to memory. `HALT` stops the processor. Initially PC = 200. Table 1.1.14 shows each step.

Step	Phase	What happens	PC	IR	R1	[207]
0	Start	The program and data are in memory.	200	—	—	0
1	Fetch	The instruction at 200 is loaded into the IR; the PC advances.	201	`LOAD`	—	0
2	Decode	Operation LOAD, destination R1, address 205.	201	`LOAD`	—	0
3	Execute	[205] = 102 is read and written to R1.	201	`LOAD`	102	0
4	Fetch	The instruction at 201 is fetched.	202	`ADD`	102	0
5	Decode	Operation ADD; R1 and [206] are to be added.	202	`ADD`	102	0
6	Execute	The ALU computes 102 + 25 = 127; the result goes to R1.	202	`ADD`	127	0
7	Fetch	The STORE at 202 is fetched.	203	`STORE`	127	0
8	Decode	R1 is to be written to address 207 (control bus: "write").	203	`STORE`	127	0
9	Execute	[207] ← 127. The result is now in memory.	203	`STORE`	127	127
10	Fetch	The HALT at 203 is fetched.	204	`HALT`	127	127
11	Decode + Execute	HALT: the processor stops.	204	`HALT`	127	127

Table 1.1.14: Fetch–decode–execute steps on the toy computer.

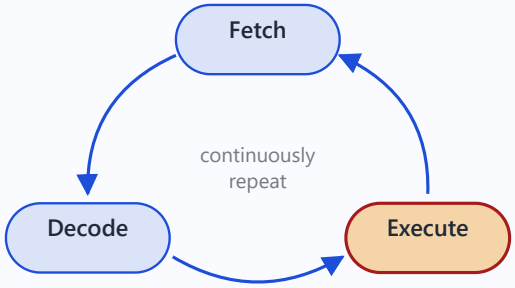


Figure 1.1.18: The state of the cycle at step 6: the execute phase of the ADD instruction is active. The ALU computes 102 + 25 = 127.

Three observations from the trace are important. First, the PC advances immediately in each Fetch step; while an instruction is being executed, the PC already points to the next one. When the program ends, the PC is 204, because it advanced once more at the moment HALT was fetched. Second, in the execute phase the bus is used in both directions. LOAD and ADD read from memory and STORE writes to memory; the "read/write" signal on the control bus decides which. Third, the `...` at address 204 never runs. Since HALT stops the processor, the cell the PC points to is never fetched again.

The Java line `total = a + b;` also ends up as instructions of this kind. The compiler produces a few machine instructions that take `a` and `b` from memory or registers, add them, and write the result to the location of `total`. The instruction sets of real processors contain hundreds of instructions, but each of them runs through the same fetch–decode–execute cycle.

Example 1.1.10

Suppose the following program, starting at address 200, runs on the same toy computer: 200: `LOAD R1, [206]` ; 201: `JUMP 205` ; 202–204: other instructions; 205: `HALT` ; 206: 40. What are R1 and PC when the program ends?

Fetch 200: the PC becomes 201; in execution, [206] = 40 is written to R1. Fetch 201: the PC first becomes 202; in execution, JUMP writes 205 into the PC. This jump uses an absolute address. Fetch 205: the PC becomes 206; HALT stops the processor. In the end R1 = 40 and PC = 206. The instructions at 202–204 were never fetched.

1.1.8 The von Neumann architecture

1.1.8.1 Architecture and the bottleneck

Definition 1.1.11

The **von Neumann architecture** is the hardware blueprint of the stored-program idea: a processor, memory, input/output units, and the bus connecting them. Instructions and data sit in the same memory and travel over the same path.

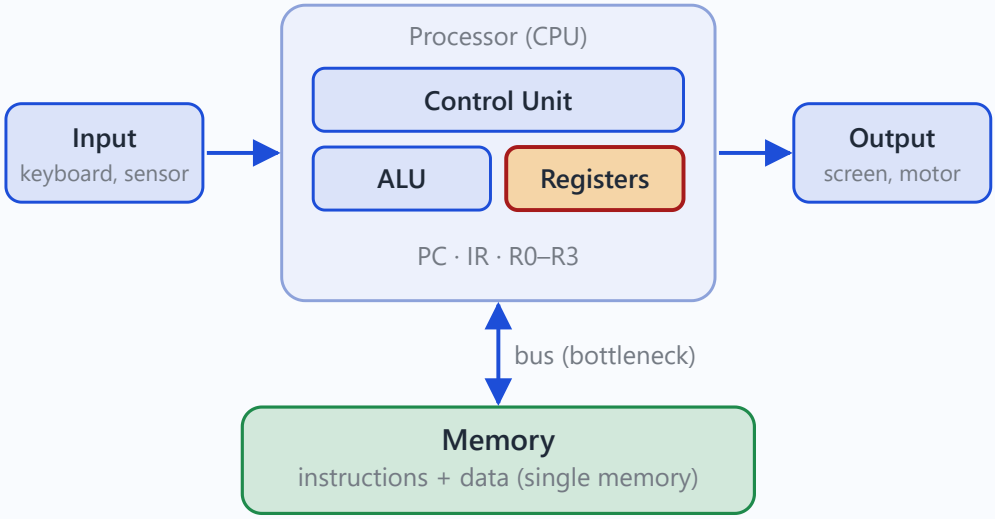


Figure 1.1.19: The von Neumann architecture: input, the processor (control unit, ALU, registers), output, and a memory in which instructions and data sit together. There is a single path between the processor and memory.

In this architecture every instruction is represented by an **opcode**. The opcode is the part of the instruction's bits that says which operation is to be performed: addition, loading, jumping? The remaining bits of the instruction give the operands, that is, which registers or which address are to be used.

This design has a cost. The single path between the processor and memory must carry both instructions and data. Because the processor runs much faster than memory, it often waits for an instruction or datum to arrive over this path. This limitation is called the **von Neumann bottleneck**. Caches exist for this reason: by keeping frequently used instructions and data next to the processor, they reduce the number of trips down the slow path.

For comparison, in the **Harvard architecture** the instruction memory and the data memory are separate, each with its own path. The processor can fetch an instruction and read a datum at the same time. Many microcontrollers, such as the AVR on Arduino boards, use this structure: the program is in

flash memory and the data in SRAM. Modern desktop processors use a hybrid. Main memory is single (von Neumann), but the L1 cache inside the processor is split in two, for instructions and for data (Harvard).

1.1.8.2 Encoding instructions

What does it mean for instructions to be numbers? Let us think again about a toy computer: an 8-bit machine with 4 registers (R0–R3), 256 memory cells of 8 bits each, and only 4 instructions. The instructions are addition (00), subtraction (01), multiplication (10) and division (11). Each instruction is an 8-bit number, and the bits are divided into four fields.

- The first 2 bits are the **opcode**. 2 bits are enough to tell 4 instructions apart, because $2^2 = 4$.
- The next 2 bits are the **destination register**: the register the result is written to. 4 registers need 2 bits.
- The last 4 bits are the two **operand registers**, 2 bits each.

The instruction "add the contents of R0 and R2 and write the result to R3" is written with its fields as *0 3 0 2*: opcode 0 (addition), destination 3, operands 0 and 2. Converting each field to binary with 2 bits and putting them side by side gives `00 11 00 10` = `00110010`.



Figure 1.1.20: The fields of the 8-bit instruction 00110010: opcode 00 (addition), destination 11 (R3), first operand 00 (R0), second operand 10 (R2).

The processor's "decode" step is to extract these bit fields from the number again. The same can be done in Java with place-value division and remainders.

```
int instr = 0b00110010;
int opcode = instr / 64;     // first 2 bits
int dest = (instr / 16) % 4;   // next 2 bits
int src1 = (instr / 4) % 4;  // next 2 bits
int src2 = instr % 4;        // last 2 bits
System.out.println(instr);
System.out.println(opcode + " " + dest
    + " " + src1 + " " + src2);
```

```
50
0 3 0 2
```

The logic of the code is easy to see with an analogy in decimal. To find the hundreds digit of 4728, first divide by 100 (47), then take the remainder modulo 10 (7). Binary works the same way. Dividing by 2^k drops the k bits on the right; taking the remainder modulo 4 gives the last 2 bits, since $4 = 2^2$. Each field is extracted in turn with these two operations.

1.1.8.3 Code trace: decoding an instruction

Table 1.1.15 traces each line of the code together with its binary form.

Line	Intermediate value	Result	Explanation
`instr = 0b00110010`	—	50	*32 + 16 + 2 = 50*
`opcode = instr / 64`	—	0	*64 = 2^6*; the 6 bits on the right are dropped, leaving `00`: addition.
`dest = (instr / 16) % 4`	*50 / 16 = 3* (`0011`)	3	The 4 bits on the right are dropped; % 4 takes the last 2 bits: `11` = R3.
`src1 = (instr / 4) % 4`	*50 / 4 = 12* (`001100`)	0	The 2 bits on the right are dropped; *12 % 4 = 0*: R0.
`src2 = instr % 4`	—	2	The last 2 bits are taken directly: `10` = R2.

Table 1.1.15: Decoding the fields for `instr = 0b00110010`.

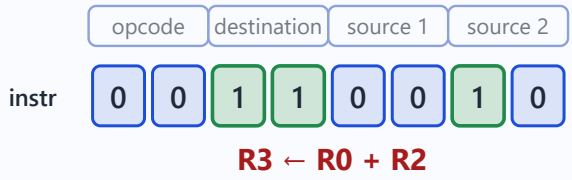


Figure 1.1.21: The decoded fields: 0 3 0 2. To the processor, the number 50 is the instruction "R3 ← R0 + R2".

The program first prints `50`, then `0 3 0 2`. Looked at as an `int`, 50 is an ordinary integer; to this toy processor it is the instruction "R3 ← R0 + R2". The meaning of a number depends on who uses it and how.

In real Java code, bit fields are usually extracted with **bitwise operators** rather than division. `instr >> 4` shifts the number 4 bits to the right and, for positive numbers, gives the same result as `instr / 16`. `& 3` masks out and keeps the last two bits, doing the job of `% 4`. The destination field is thus written `(instr >> 4) & 3`. This notation is closer to what the hardware actually does: the control unit does not divide, it just selects the relevant wires.

Example 1.1.11

What does the instruction `10011100` do on the toy computer?

The bits are split into 2-bit fields: `10 01 11 00`. The opcode 10 is multiplication. The destination is 01, that is, R1. The operands are 11 (R3) and 00 (R0). The instruction means **R1 ← R3 × R0**. To check with Java code, take the numeric value of the instruction, *128 + 16 + 8 + 4 = 156*: *156 / 64 = 2*; *156 / 16 = 9* and *9 % 4 = 1*; *156 / 4 = 39* and *39 % 4 = 3*; *156 % 4 = 0*. The fields are 2 1 3 0.

1.1.8.4 Assembly language

Writing instructions as bit strings like `00110010` is very hard for people. **Assembly language** gives each machine instruction a memorable abbreviation such as `ADD` or `MOV`. A program called an **assembler** translates text written in assembly into machine code.

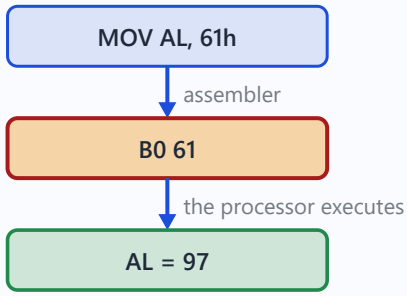


Figure 1.1.22: The assembler translates the instruction `MOV AL, 61h` into the machine code B0 61; when the processor executes it, 97 is written to the AL register.

- Assembly language is specific to a particular processor architecture. Code written for x86 does not run on an ARM processor, because the instructions and encodings of the two architectures differ.
- Each line generally corresponds to a single machine instruction. A single line of Java, on the other hand, often turns into many machine instructions.
- It is important for understanding how the computer works at a low level.
- It is used in performance tuning and in direct interaction with hardware: device drivers, embedded systems, and some parts of operating system kernels.

x86 instruction	Meaning	Machine code
`MOV AL, 61h`	Move the value 61h (= 97) into register AL	`B0 61`
`MOV AX, BX`	Copy the value in BX to AX	`66 89 D8`
`ADD EAX, 10`	Add 10 to EAX	`83 C0 0A`
`XOR EAX, EAX`	Clear EAX (x XOR x = 0)	`31 C0`

Table 1.1.16: Some x86 instructions and their machine code (hexadecimal).

The codes in Table 1.1.16 are the real-processor counterpart of the idea in the toy computer. In the code `B0 61`, B0 is the opcode for "load a constant into register AL", and 61 is the constant to load. In `ADD EAX, 10`, the last byte is 0A = 10. x86 instructions are not a fixed 8 bits as in the toy computer; they vary from 1 to 15 bytes. The 66 byte at the start of the `MOV AX, BX` code is a prefix saying that the instruction works with 16-bit rather than 32-bit registers. `XOR EAX, EAX` is a common way to clear a register. The XOR of a number with itself gives 0 in every bit. Moreover, this instruction is encoded shorter than `MOV EAX, 0`.

The 61h in the first row means hexadecimal 61 (the `h` suffix in assembly, the `0x` prefix in Java). 61h = 0x61 = 97, and 97 is also the ASCII `'a'`. Is a byte in memory a number, a letter or an instruction? The byte itself does not say; how the program uses it decides. This sentence is the common conclusion of all the layers in these notes.

Java's place among these layers also becomes clear here. The programmer writes in a high-level language, Java. `javac` translates this code to bytecode. Bytecode is the instruction set not of a real processor but of the JVM, and it is the same on every processor. While running, the JVM translates the bytecode into the machine instructions, at the assembly level, of that particular machine. Java's principle of "write once, run anywhere" is possible thanks to this intermediate layer.

1.1.9 Common mistakes

The most common consequences of the topics in these notes show up in Java as lines the compiler rejects, or as programs that compile and run wrongly. In the list below, each mistake is given together with the compiler message or the program's actual output. The messages are from the JDK 21 compiler.

Mistake	Code	Result	Cause and fix
Out-of-range literal	`byte b = 200;`	`error: incompatible types: possible lossy conversion from int to byte`	`byte` goes up to 127 at most. 200 is an `int` literal and does not fit; if you really want it, write `(byte) 200` and the value becomes −56.
Forgetting the `L` suffix	`long x = 3000000000;`	`error: integer number too large`	An integer literal without a suffix is an `int`, and an `int` cannot exceed 2,147,483,647. The fix is `3000000000L`.
Confusing a character with a string	`char c = "A";`	`error: incompatible types: String cannot be converted to char`	Double quotes produce a `String`. A character is written with single quotes: `'A'`.
Leading zero	`int x = 010;` `int y = 08;`	`x = 8` `error: ';' expected`	A leading 0 means octal; 010 = 8. Since there is no digit 8 in octal, the second line does not compile.
Not noticing overflow	`Integer.MAX_VALUE + 1`	`-2147483648`	Integer overflow raises no error; it wraps around. Large products need `long`, and the product must start out as a `long`: `1920L * 1080`. For checking, use `Math.addExact`.

Mistake	Code	Result	Cause and fix
`==` on fractions	`0.1 + 0.2 == 0.3`	`false`	Floating-point values are approximate. Look at whether the difference is small: `Math.abs(a - b) < 1e-9`.
Taking `length()` for the number of characters	`"😜".length()`	`2`	`length()` counts UTF-16 units. Use `codePointCount` for code points and `getBytes(...).length` for bytes.
Taking a `char` sum for a string	`System.out.println('a' + 'b');`	`195`	The sum of two `char`s is an `int`. To concatenate, use `"" + 'a' + 'b'` or `String.valueOf`.

Table 1.1.17: Common mistakes and their causes.

The compiler catches the mistakes in the first four rows of the list, except for `010`, and the program never runs. These are annoying but harmless, because the line and cause of the error are seen immediately. The code for `010` and in the last four rows, however, compiles and silently produces wrong results, which makes it more dangerous. An overflowing counter, a floating-point equality or a wrongly computed string length is noticed only when the program behaves unexpectedly somewhere else. The common source of these mistakes is forgetting how a value is represented in memory.

1.1.10 Summary

- **Components.** The CPU computes and consists of the ALU, the control unit and registers. RAM is temporary and disk permanent memory. The bus carries address, data and control signals; an address bus with n lines selects 2^n addresses.
- **Binary numbers.** A bit is 0 or 1; 8 bits make 1 byte. With d digits and base b, the numbers 0 … $b^d - 1$ can be written. To convert decimal to binary, divide by 2 and read the remainders from bottom to top. A hexadecimal digit is 4 bits and an octal digit is 3 bits.
- **Bases in Java.** `0b` marks a binary literal, `0` an octal one and `0x` a hexadecimal one. `Integer.toBinaryString`, `toHexString` and `parseInt(s, radix)` convert.
- **Negative numbers.** Two's complement: flip the bits and add 1. The weight of the leftmost bit is negative. With 8 bits the range is −128 … 127. If the range is exceeded, Java overflows silently and the value wraps around.
- **Fractional numbers.** Stored as fraction × base^{exponent}. By IEEE 754, `float` is 32 bits and `double` is 64 bits. Values are approximate: `0.1 + 0.2 != 0.3`.
- **Text.** ASCII is 7 bits and 128 characters. Unicode gives every character a code point (U+0000–U+10FFFF). UTF-8 writes a code point in 1–4 bytes and is compatible with ASCII. Java's `char` is a 16-bit UTF-16 unit; `length()` gives the number of `char`s.
- **Structured information.** An image is an array of RGB pixels, sound is pressure samples, and video is a sequence of frames. One minute of uncompressed Full HD video takes about 11 GB.

- **Stored program.** Instructions also sit in memory as numbers. The PC holds the next address. The processor repeats the fetch–decode–execute cycle continuously; jump instructions change the order.
- **The von Neumann architecture.** Instructions and data share the same memory and the same path; the single path creates a bottleneck, which caches ease. In the Harvard architecture, instruction and data memories are separate.
- **Levels.** High-level language → assembly → machine code (bits). Assembly is processor-specific, and each line is usually a single instruction. Whether a byte is a number, a letter or an instruction is decided by the program that uses it.

1.1.11 Key terms

CPU RAM bit byte binary number system positional value hexadecimal carry two's complement overflow floating-point number character encoding ASCII Unicode code point UTF-8 surrogate pair stored-program computer memory hierarchy cache bus register program counter fetch–decode–execute cycle von Neumann architecture opcode assembly language

---

1.1.12 Test Yourself

1. How is 100 written as an 8-bit binary number and in hexadecimal?

   Answer

   The powers that fit in 100 are 64 and 32, leaving 4. *100 = 64 + 32 + 4*, which is `01100100`. Groups of four give `0110 0100` = 6 and 4, so the hexadecimal value is `0x64`.

2. What is the bit pattern `11010110` as an unsigned number and as an 8-bit two's complement number?

   Answer

   Unsigned: *128 + 64 + 16 + 4 + 2 = 214*. Signed: take the weight of the leftmost bit as −128, *−128 + 64 + 16 + 4 + 2 = −42*; or *214 − 256 = −42*.

3. How are −1 and −128 written in 8-bit two's complement? For `byte m = -128;`, why does `(byte)`

`(-m)` again give −128?

Answer

−1 = `11111111` (1 = `00000001`, inverted `11111110`, +1). −128 = `10000000`. The number 128 does not exist in the range of `byte`. Flipping the bits of −128 gives `01111111`, and adding 1 gives `10000000` again. In Java, `-m` is first computed as the `int` 128, but the cast to `byte` keeps only the last 8 bits and the result is −128.

4. What do these lines print? `System.out.println((byte) 130);`, `System.out.println((char) ('A' + 2));`, `System.out.println('A' + 2);`

Answer

−126, `C` and 67. 130 = `10000010`, read as signed, is *130 − 256 = −126*. `'A' + 2` is an `int` (67); with the `(char)` cast, the character numbered 67, `C`, is printed; without the cast the number itself is printed.

5. For the string `"Işık"` ("light"), what are `length()` and the number of UTF-8 bytes?

Answer

`length()` is 4; all four characters are below U+FFFF. In UTF-8, I (U+0049) is 1 byte, ş (U+015F) is 2 bytes, ı (U+0131) is 2 bytes and k is 1 byte: *1 + 2 + 2 + 1 = 6*.

6. What does `System.out.println(0x2A + 012 + 0b11);` print?

Answer

55. `0x2A` = *2·16 + 10 = 42*, `012` is octal, so *1·8 + 2 = 10*, and `0b11` = 3. The total is *42 + 10 + 3 = 55*; the result is printed in decimal.

7. On the toy computer (2-bit opcode: 00 add, 01 subtract, 10 multiply, 11 divide; 2-bit destination; two 2-bit operands), which 8-bit number is the instruction "R1 ← R2 − R3"? Which fields come out when it is decoded with the Java code from the notes?

Answer

`01 01 10 11` = `01011011` = 91. Decoding: *91 / 64 = 1*; *91 / 16 = 5*, *5 % 4 = 1*; *91 / 4 = 22*, *22 % 4 = 2*; *91 % 4 = 3*. The fields are 1 1 2 3.

8. A processor has a 20-line address bus, and each address selects one byte. What is the most

memory it can address?

Answer

$2^{20} = 1{,}048{,}576$ bytes, that is, 1 MiB. The Intel 8086 processor had exactly this address bus width.

9. Telephone-quality sound is recorded at 8000 samples per second, 1 byte per sample and a single channel (mono). How many bytes is one minute of uncompressed speech? If the computation is done with `int`, does overflow occur?

Answer

$8000 \cdot 1 \cdot 1 \cdot 60 = 480{,}000$ bytes. This is far smaller than `Integer.MAX_VALUE`, so there is no overflow. For CD audio the same computation gives 10,584,000 bytes, which also fits in an `int`. Overflow starts with products that run into the billions, like video.

10. In Java, the expression `0.1f + 0.2f == 0.3f` gives `true`. Does this show that equality comparison is safe with `float`?

Answer

No. `float`'s coarser rounding happened to carry the errors to the same number for these three values. The same sum gives `false` with `double`, and with other numbers `float` fails too. Floating-point values must always be compared with a tolerance.

1.1.13 Further reading

- C. Petzold, *Code: The Hidden Language of Computer Hardware and Software* (2nd ed., Microsoft Press, 2022): starting from relays and logic gates, it builds the adder, the memory and a simple processor step by step; it is the most readable long-form version of the layered approach taken in these notes.
- R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective* (3rd ed., Pearson, 2016), Chapter 2: treats unsigned and two's complement integers, overflow, bitwise operations and IEEE 754 floating point from the programmer's viewpoint, with many exercises.
- D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic", *ACM Computing Surveys* 23(1), 1991: the classic paper explaining in detail where rounding error comes from and the design rationale of the IEEE 754 standard.
- D. A. Patterson and J. L. Hennessy, *Computer Organization and Design* (RISC-V edition, 2nd ed., 2020), Chapters 2 and 5: explains instruction encoding on a real instruction set and the memory hierarchy with cache details.

1.1.14 References

- D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, RISC-V edition, 2nd ed., Morgan Kaufmann, 2020.
- C. Petzold, *Code: The Hidden Language of Computer Hardware and Software*, 2nd ed., Microsoft Press, 2022.
- A. B. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019 (Chapter 1).
- Oracle, *The Java Tutorials*: "Primitive Data Types" and "Characters".
- J. Gosling et al., *The Java Language Specification*, Java SE 21, §3.10 (Literals) and §4.2 (Primitive Types).
- The Unicode Consortium, *The Unicode Standard*, unicode.org; IEEE 754-2019, *Standard for Floating-Point Arithmetic*.
- J. von Neumann, *First Draft of a Report on the EDVAC*, 1945.
- R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3rd ed., Pearson, 2016.
- D. Goldberg, “What Every Computer Scientist Should Know About Floating-Point Arithmetic”, *ACM Computing Surveys*, 23(1), 5–48, 1991.

PART I · CHAPTER 1.2

Computation and Computational Thinking

A pocket calculator only adds and multiplies. A laptop, on the same day, plays music, simulates circuits and compiles Java. The difference comes less from the power of the hardware than from the fact that the instructions given to the machine can be changed. These notes take up the basic question of what a computer does: what is computation, how is a problem put into a form a computer can solve, and where does that road end? The first sections define programming, computation and computer science. Then the four skills of computational thinking, namely decomposition, pattern recognition, abstraction and algorithm design, are worked through on a single example. The history section runs from punched cards to the stored-program computer. The Turing machine gives an exact definition of computation and shows that not every problem is solvable. The last sections separate declarative from imperative knowledge using the square root and GCD examples, and present algorithms, problem specification, and sequential, conditional and repetitive operations with Java code.

LEARNING OUTCOMES

Basic concepts
Define programming, computation and computer science; explain that computer science is more than technology.

Computational thinking
Apply the skills of decomposition, pattern recognition, abstraction and algorithm design to a problem in turn.

Historical development
Tell fixed-program from stored-program computers; explain the place of Babbage, Lovelace, the ABC, Colossus, ENIAC, EDVAC and the von Neumann architecture.

Turing machine
Run a simple Turing machine step by step with its tape, tape head and rule table; explain the concepts of computability and the halting problem.

Kinds of knowledge
Tell declarative (what?) from imperative (how?) knowledge using the square root and GCD examples; trace the Babylonian method and Euclid's algorithm in Java.

Algorithms and operations
List the properties of an algorithm and identify the input and output of a problem; recognize sequential, conditional and repetitive operations, and explain the relationship between `repeat … until` and `do … while`.

1.2.1 Programming and computation

A computer is a helper that works very fast but knows nothing by itself. It performs billions of operations a second, yet it cannot decide on its own which operation to do in which order. That decision is given to it as a written list of instructions. The list is called a **program**, and the act of writing the list is **programming**. The job a program carries out is, most of the time, a **computation**: data is taken in, processed and a result is produced. This section defines the two concepts separately and shows how they come together in a small Java program.

1.2.1.1 Programming

Definition 1.2.1

Programming is the activity of writing instructions that tell a computer, step by step, what to do. The text formed by these instructions is called a **program**. The instructions are written in a **programming language**; Java, Python and C are examples.

A programming language is a bridge between humans and machines. It must be readable for people and exact for machines. In everyday speech, saying "wait a bit" or "pick a suitable number" is enough; the listener fills in the gaps with their own interpretation. A computer makes no such interpretation. If how long to wait or which number to pick is not written out explicitly, it either reports an error or does something wrong. That is why the rules of programming languages are strict: if a single semicolon is missing, a Java program does not compile.

A programmed computer can carry out complex tasks. Smartphones, tablets, laptops, washing machines and cars all work thanks to the programs running inside them. Online shopping, news, communication and education are also products of programs. Computers have entered every field, from home to workplace and from entertainment to education.

One benefit of learning programming also shows in how you think. To get a computer to do a job, you first have to understand that job exactly. Behind a short request like "take the average" lie questions such as how many numbers there are, where they come from and how the result will be written. A

person who learns to ask and answer such questions also thinks more systematically when solving problems.

1.2.1.2 The role of programming

Programming is a tool for getting computers to perform the tasks we want. Programmers make computers more useful and efficient in every area of life. Three areas illustrate this role well:

- **Automation:** Smart thermostats measure the temperature and switch the heater on or off. Lighting and security systems come into action by themselves under certain conditions. Robots in factories do the same weld or assembly thousands of times with the same accuracy.
- **Health:** A pacemaker continuously measures the rhythm of the heart and gives a warning when needed. An MRI scanner produces an image from measured signals; this conversion is a computation from start to finish. Hospital information systems and appointment systems also run on programs.
- **Education:** Online course platforms, teaching software, and exam and grading systems are products of programmed computers.

Electrical and electronics engineering touches all of these areas. Embedded software loaded onto a microcontroller, a digital filter that cleans up a signal from a sensor, or a control loop that sets the speed of a motor are all written through programming. In such systems the boundary between circuit design and software grows ever thinner.

1.2.1.3 Computation

Definition 1.2.2

Computation is the process by which a computer or computing device processes data to perform a task. A computation takes data as input, processes it with mathematical operations and logical decisions, and produces a result.

Computation is the basic function of computers. Even drawing a letter on the screen is a computation: the shape, size and position of the letter are turned into numbers, and from these numbers it is computed which pixels will be painted which color. Computation involves three kinds of work. **Mathematical operations** are operations such as addition and multiplication. **Logical decisions** check whether a condition is true. **Data processing** covers general jobs such as sorting, searching and formatting data.



Figure 1.2.1: The three stages of a computation. Data (7 and 3) is taken in, passes through operations such as addition, multiplication and a decision, and a result (10 and 21) is produced.

The pattern seen in Figure 1.2.1 is the same in every computation. What changes is the kind of data and the complexity of the operation. A few examples:

- **Data analysis:** From measurements taken from a temperature sensor over a day, the average, the maximum and the trend are computed; the result is drawn as a chart.
- **Simulation:** A mathematical model of a circuit or of the weather is built. The computer advances the model in small time steps and predicts how the system will behave.
- **Game development:** In every frame, the positions of objects are updated by the rules of physics, collisions are checked, and the computer-controlled characters make decisions.

The relationship between programming and computation can be summed up like this: computation is the work being done, and programming is writing the recipe for that work. A well-written program has the same computation done in fewer steps, with less memory and with fewer errors. That is why programming makes computation more effective and efficient.

1.2.1.4 The building blocks of computation

Computation has two basic building blocks. The first is **mathematics**: operations such as addition, subtraction, multiplication and division. The second is **logic**: decisions are made by evaluating logical expressions. The sentence "If the weather is sunny, go outside" is a logical operation. First the answer to the question "is the weather sunny?" is found; the answer is true or false. Then an action is chosen according to that answer.

The following Java program uses the two building blocks together. It computes the sum and product of two numbers, then decides according to a condition.

```java
public class Computation {
    public static void main(String[] args) {
        int a = 7, b = 3;
        int sum = a + b;     // math
        int product = a * b;     // math
        boolean sunny = true; // logic
        System.out.println("Sum: " + sum);
        System.out.println("Product: " + product);
        if (sunny) {          // decision
            System.out.println("Go outside");
        }
    }
}
```

When run, the program produces the following output:

```
Sum: 10
Product: 21
Go outside
```

The program runs, line by line, as follows:

- The line `public class Computation` defines the program's class. In Java all code is written inside a class, and the file name must match the class name: `Computation.java`.
- The `main` method is the entry point of the program. When the Java virtual machine starts the program, it calls this method first.
- The line `int a = 7, b = 3;` defines two integer variables and gives them their initial values. These are the inputs of the computation.
- The `sum` and `product` lines use the mathematical building block. The expression on the right is evaluated and the result is stored in the variable on the left.
- The line `boolean sunny = true;` prepares the data for the logical building block. A variable of type `boolean` can hold only one of two values: `true` or `false`.
- The two `println` lines print the results. In the expression `"Sum: " + sum`, the `+` sign appends the number to the text; this is called string concatenation.
- The line `if (sunny)` makes the decision. Since the condition is `true`, the block inside the braces runs and "Go outside" is printed. Had the condition been `false`, the block would be skipped and the program's output would be only two lines.

1.2.1.5 Code trace: the Computation program

The most reliable way to understand what a program does is to trace it by hand. A **code trace** means reading the program as if executing it line by line, and recording at every step the values of the variables and what is written to the screen. Table 1.2.1 is the trace of the `Computation` program. Line numbers are counted from the first line of the code: line 1 is the class header, and line 3 is `int a = 7, b = 3;`.

Line	a	b	sum	product	sunny	Output (new line)
3	7	3				
4	7	3	**10**			
5	7	3	10	**21**		
6	7	3	10	21	**true**	
7	7	3	10	21	true	Sum: 10
8	7	3	10	21	true	Product: 21
9	7	3	10	21	true	(condition true)
10	7	3	10	21	true	Go outside

Table 1.2.1: Trace of the Computation program. An empty cell means the variable is not yet defined.

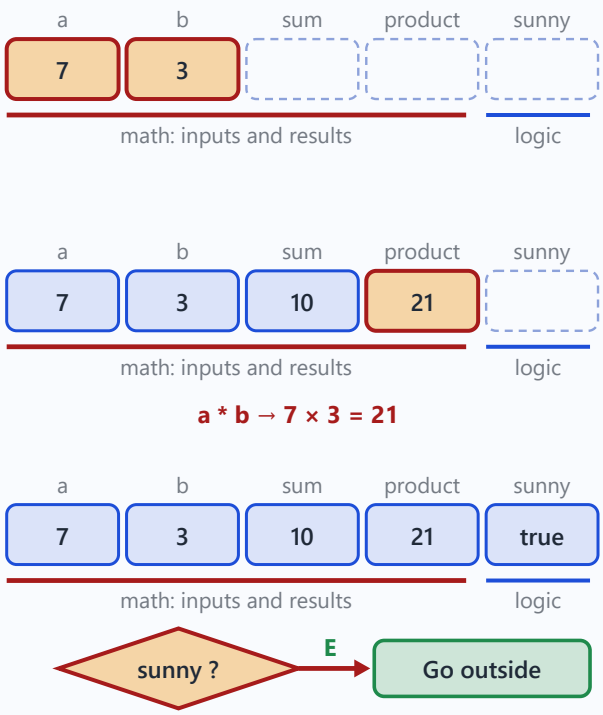


Figure 1.2.2: Three moments from the code trace. (a) Line 3: the inputs `a` = 7 and `b` = 3 are stored; the other boxes are still empty. (b) Line 5: `a * b` = 21 is computed. (c) Line 10: since `sunny` is true, the "Yes" branch of the decision box is chosen and "Go outside" is printed.

Two details in the trace deserve attention. First, once a variable receives a value it keeps it until a new assignment comes; in the table, `a` and `b` do not change from start to finish. Second, nothing is printed at line 9. Only the condition is evaluated on this line and the path to follow is determined. The result of the decision shows up on the next line. In this way the program shows the three stages of computation in order: data was taken in, processed, and a result produced.

Common mistake

Forgetting the order of operations in string concatenation. The line `System.out.println("Sum: " + a + b);` prints "Sum: 73", not "Sum: 10". Java evaluates the expression from left to right: first `"Sum: " + a` is concatenated, forming the string "Sum: 7", and then `b` is appended to it. To add the numbers first, parentheses are needed: `"Sum: " + (a + b)`.

1.2.2 Computer science

The name "computer science" can be misleading. The name suggests that the field is about the machine called the computer. Yet the real subject of the field is not the machine but the problems the machine solves, and the ways of solving those problems.

1.2.2.1 More than technology

Definition 1.2.3

Computer science is the discipline that studies how information is processed, stored and transmitted, and investigates ways of defining and analyzing problems and producing effective solutions to them.

Computer science does not consist merely of technical knowledge about computers; it is also a discipline concerned with problem-solving skills. Technology is a tool in this discipline. The main aim is to solve problems and to understand information better. A frequently quoted analogy makes this clear: computer science is about computers as much as astronomy is about telescopes. The telescope is a tool for studying the sky; the computer is a tool for solving problems.

This view has a practical consequence. The details of a programming language or a development environment change in a few years. The ability to break a problem into pieces, choose the right data structure and test the correctness of a solution does not change. In this course you learn Java, but the way of thinking behind Java carries over to other languages.

1.2.2.2 Three pillars: logic, problem solving, creativity

Computer science rests on three skills:

- **Logic:** The ability to evaluate data and draw correct conclusions. It forms the foundation of writing programs. The condition of an `if` statement, when a loop stops, whether two conditions are joined with `&&` or `||` are all questions of logic.
- **Problem solving:** The skill of defining and solving complex problems. Every problem is solved with a logical sequence of steps. Finding that sequence is the hardest and most instructive part of programming.
- **Creativity:** Makes new and innovative solutions possible. The same problem can have many different solutions. The GCD of two numbers can be found by trying all the numbers one by one, or by Euclid's algorithm; both give the same result, but for large numbers one is much faster than the other.

These three skills are not taught separately; they develop together as you program. Logic is used when debugging a program, problem solving when designing its parts, and creativity when looking for another way at a point where you are stuck.

1.2.2.3 Technology and social responsibility

Computer science leads us to think about how technology can help people and contribute to solving social problems. It helps us understand and improve the world. It also stresses the responsibility of putting technology at the service of humanity. That is why the field is interesting not only to those concerned with technology but to anyone who wants to develop problem-solving and thinking skills.

The areas of application are very wide: medicine, economics, astronomy, biology, energy, transport and disaster management are only a few. After an earthquake, damage assessment is sped up by programs that process satellite images. Load balancing in an electrical grid is done with models that forecast consumption.

This breadth comes at a price: the programs we write affect real people. A bug in hospital software can lead to a wrong drug dose, and failing to protect personal data can cause serious harm. A programmer's responsibility does not end when the code compiles; it includes making sure the code works correctly and does not harm its users.

1.2.2.4 The topics of computer science

Computer science aims to understand, model and analyze reality. Algorithms and code are used to solve problems. Three broad themes stand out within the field:

- **Abstraction:** Hiding the unnecessary details of a complex system and keeping its essence. A metro map is a good example. The map does not show streets, buildings or real distances; it shows only the stations and the lines. That is what the passenger needs to know: which station to board at and where to change.

- **Problem solving:** Breaking the problem into parts, solving each part step by step, and combining the solutions.
- **Art and design:** Computer science is not limited to logic and analytical thought. Digital media, electronic music, games and animation are also products of the field.

Abstraction and problem solving are also two of the legs of computational thinking. Pattern recognition and algorithm design complete the pair, and the four skills together form a working method.

1.2.3 Computational thinking

The hardest part of writing a program is often not remembering the rules of Java. The real difficulty comes before the first line of code is written. What does the problem ask, and what parts does it consist of? What information is needed and what is unnecessary? In what order will the parts be solved? The habit of looking systematically for answers to these questions is called **computational thinking**.

1.2.3.1 Definition

> **Definition 1.2.4**
>
> **Computational thinking** is a way of expressing a problem and its solution so that a person or a computer can carry it out step by step. It rests on four skills: **decomposition**, **pattern recognition**, **abstraction** and **algorithm design**.

The term was popularized in 2006 by a short article by Jeannette Wing. Wing argued that this way of thinking is needed not only by computer scientists but by everyone, and counted it as a basic skill like reading, writing and arithmetic. The four-skill breakdown is a framework widely used in schools and universities to teach this idea.

Computational thinking does not mean "thinking like a computer". A computer does not think; it only executes instructions. Computational thinking is a person's ability to make a problem explainable to a computer. The same skill is useful without a computer. Writing a recipe, deciding the order of measurements to test a circuit, or making a study plan for an exam can be organized with the same four steps.

Seeing the four skills on a single problem is more instructive than defining them one by one. The example problem throughout this section is: *finding the grade average of the students in a class*.

1.2.3.2 Decomposition

Definition 1.2.5

Decomposition is breaking a large and complex problem into small subproblems that can each be solved separately.

The request "find the grade average" looks like a single step, but for a computer it consists of four separate jobs:

1. Read the grades.
2. Add up the grades.
3. Divide the sum by the number of grades.
4. Write the result.

Each subproblem is easy on its own. An input statement is enough for reading, a loop for summing, a single operation for dividing, and `println` for writing. This is where the power of decomposition comes from: a hard problem turns into a combination of easy problems.

Decomposition is not done once and left. If a subproblem is still large, it is split too. The step "read the grades" can be broken down further depending on whether the grades come from the keyboard, from a file or from a database. In Java each subproblem often corresponds to a separate method. Large programs are built this way: classes and methods are the traces decomposition leaves in code.

The counterpart of decomposition in electronics is the block diagram. When designing an audio amplifier, you first separate blocks such as the preamplifier, the tone control and the power stage; each block is designed and tested separately, then combined. The methods in a program can likewise be written and tried out one by one.

1.2.3.3 Pattern recognition

Definition 1.2.6

Pattern recognition is seeing similarities among subproblems, or between a problem and other known problems. If a job is found to repeat in the same form, a single solution is applied to all of it.

In the grade average example the pattern is clear. Reading the first grade and adding it to the sum is the same job as reading the second grade and adding it to the sum; only the grade differs. For a class of thirty students, there is no need to write this step thirty times separately. The repeating step is written once and put in a **loop**. In programming, pattern recognition often leads to the idea "there is a loop here".

Patterns are looked for not only inside a single problem but also between problems. Averaging grades and averaging a sensor's temperatures over a day are the same problem: take the sum of a list of numbers and divide it by the number of elements. A solution written for one fits the other with a small change. The Babylonian method and Euclid's algorithm also have a repeating step: one improves a guess, the other computes a remainder. Both repeat the step until a certain condition is met.

Pattern recognition grows with experience. A programmer who has solved many problems can say of a new one, "this is a find-the-maximum problem" or "this is a merge-two-lists problem". Reducing a problem to a recognized pattern is much faster than finding the solution from scratch.

1.2.3.4 Abstraction

Definition 1.2.7

Abstraction is discarding the details that are not needed to solve a problem and keeping only the important properties.

There is a lot of information about a student: name, number, date of birth, hometown, absences, grades. Most of it is unnecessary for computing an average. The student's name or number does not affect the average. Only two pieces of information are needed: the grades and the number of grades. After abstraction, the problem stops being "the average of thirty students" and becomes "the average of a list of numbers". This is a more general problem, and its solution can be used elsewhere too.

The metro map example shows the same idea: the map is not a shrunken photograph of the real city; everything except the information the passenger needs has been thrown away. A good abstraction requires being sure that the discarded detail really is unnecessary. If the grades have different weights, for example if the midterm is 40% and the final 60%, the abstraction "the grades alone are enough" is wrong; the weights must also enter the model.

Abstraction is also the foundation of object-oriented programming. When designing a `Student` class in Java, not all the properties of a real student but only the properties the program needs are put into the class as fields. A registration system needs a name and a number, a grading system needs grades. The same real entity is abstracted differently in different programs.

1.2.3.5 Algorithm design

Definition 1.2.8

Algorithm design is writing the solution as finite, clear and ordered steps. The resulting list of steps is precise enough for a person to follow by hand or to be translated into a programming language.

The previous three skills prepare the problem; algorithm design writes the solution. For the grade average, the algorithm in the pseudocode style used in the slides is:

```
set total to 0
for each grade in grades do
    set total to total + grade
set avg to total / n
print avg
```

These steps bring together the four subproblems of decomposition, the loop found by pattern recognition, and the two pieces of information abstraction left behind (the grades and n). Every step is clear: there is no room for interpretation about what is to be done. The number of steps is finite: the loop runs once for each grade in the list and stops when the list ends.

Some details are also settled while writing an algorithm. Why does the total start at 0? Because before any grade has been added the total is zero, and each grade is added to that value. What happens if the list is empty? $n = 0$ and the division is undefined. A well-designed algorithm thinks about this case too; for example, it adds a step such as "if the list is empty, print 'no grades'".

1.2.3.6 All four skills together: the grade average

Table 1.2.2 summarizes what the four skills correspond to in the example problem.

Skill	The question it asks	Counterpart in the grade average
Decomposition	What parts does the problem consist of?	Read, add, divide, write.
Pattern recognition	Is there a repeating or familiar job?	Adding each grade to the sum is the same step: a loop.
Abstraction	What information is needed, and what is not?	Name and number are unnecessary; the grades and the number of grades are necessary.
Algorithm design	In what steps and in what order is the solution carried out?	`total ← 0`; for each grade `total ← total + grade`; `avg ← total / n`.

Table 1.2.2: The four skills of computational thinking and their counterparts in the grade average problem.

Once the algorithm is ready, translating it into Java is easy. The following program keeps the grades in an array and computes the average. The grades are 40, 75 and 90, taken from the slides' example problems.

```java
public class Average {
    public static void main(String[] args) {
        int[] grades = {40, 75, 90};   // input: grades only
        int total = 0;
        for (int grade : grades) {        // same step for each grade
            total = total + grade;
        }
        double avg = (double) total / grades.length;
        System.out.println("Average: " + avg);
    }
}
```

The program prints "Average: 68.33333333333333". Each part of the code corresponds to a skill. The `grades` array is the result of abstraction: not students but only grades are kept. The `for` loop is the repeating step that pattern recognition found. The addition and division lines are the subproblems of decomposition. And the whole code is the Java counterpart of the algorithm.

Step	`grade`	`sum`	Explanation
Start	–	0	No grade has been added yet.
Iteration 1	40	40	0 + 40
Iteration 2	75	115	40 + 75
Iteration 3	90	205	115 + 90; the list is finished and the loop exits.
Division	–	205	`avg` = 205 / 3 = 68.333…

Table 1.2.3: Trace of the Average program (grades 40, 75, 90).

The `(double)` cast in the division line is important. `total` and `grades.length` are both of type `int`. When Java divides two `int` values it gives an `int` result and discards the fractional part: 205 / 3 = 68. Once `(double) total` is written, the dividend is converted to a floating-point number and a true division is carried out. This detail is the kind of thing abstraction can overlook: the algorithm is correct on paper, but if Java's number types are not taken into account the program gives a wrong result.

Example 1.2.1

In a laboratory, the voltage across a resistor is measured every second for an hour. The task is to find the largest value of the measurements and the number of measurements exceeding 5 V. The four skills are applied in turn.

Decomposition: read the measurements; find the maximum; count those above 5 V; write the two results. **Pattern recognition:** finding the maximum and counting are each a step repeated for every measurement; both can be done in the same loop. Also, "finding the maximum" is a recognized pattern. **Abstraction:** the time of the measurement, the name of the laboratory and the color of the resistor are unnecessary; all that is needed is a list of 3600 voltage values.

Algorithm design: `maxVal ← first measurement` , `count ← 0` ; for each measurement, if the measurement is greater than `maxVal` then `maxVal ← measurement` , and if the measurement is greater than 5 then `count ← count + 1` ; at the end write the two values.

Initializing `maxVal` with the first measurement rather than 0 is a deliberate choice. If all the measurements were negative, an algorithm starting from 0 would wrongly find 0.

Common mistake

Skipping the computational thinking steps and starting to write code right away. Code written without decomposing the problem usually turns into a single long `main` method; when an error occurs, it is impossible to find which part is responsible. Code written without abstraction also deals with unnecessary data or forgets a needed piece of information (for example, the grade weights).

1.2.4 The history of the computer

Today's computer is the product not of a single invention but of a development lasting several centuries. This development revolves around two ideas. The first is having a machine do the computing: the line running from gears to vacuum tubes and then to transistors is the product of this idea. The second is being able to change what the machine does: the shift from fixed-program machines to stored-program machines is the product of this idea. The details of history are easier to remember when placed along these two lines.

1.2.4.1 What is a computer?

Definition 1.2.9

A **computer** is a device that carries out a series of computations and instructions. The instructions let the computer perform a task. A computer uses mathematical computations and logical instructions to process data.

Today's computers work with electronic circuits and represent numbers digitally. They consist of four main components:

- **Processor** (CPU, central processing unit): Executes instructions and performs computations.
- **Memory** (RAM, random access memory): Holds the running program and its data. Its content is erased when the power goes off.
- **Storage:** Units such as disks and SSDs store data permanently. Data is not lost when the computer is turned off.

- **Input/output devices:** The keyboard, mouse, screen, printer and network connection let the computer communicate with the outside world.

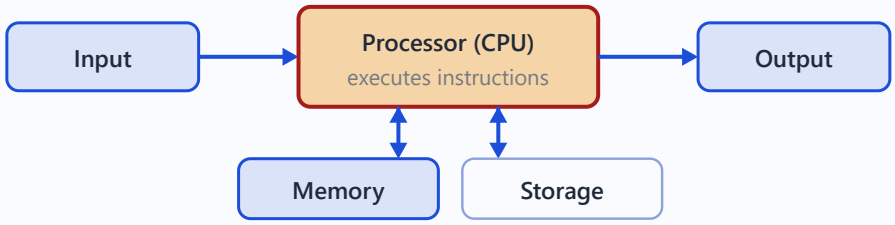


Figure 1.2.3: The components of a computer. Input data reaches the processor; the processor works together with memory and storage; the result goes to the output device.

When a Java program runs, all of these components are involved. The `.java` and `.class` files sit on disk. When the program is started, it is loaded into memory. The processor executes the instructions one by one. `println` sends the result to the screen, that is, to an output device. The difference between memory and storage matters to the programmer too: a value held in a variable is lost when the program ends, and data that must persist has to be written to a file.

1.2.4.2 Electronic and digital

Two adjectives describe today's computer: electronic and digital. They describe different things.

- **Electronic:** The computer processes data with electric current. Transistors do the switching in place of mechanical gears. Since there are no moving parts in the switching, operations are very fast.
- **Digital:** Data is represented with 0 and 1. Each 0 or 1 is a **bit**; 8 bits make a **byte**. A byte can take $2^8 = 256$ different values.

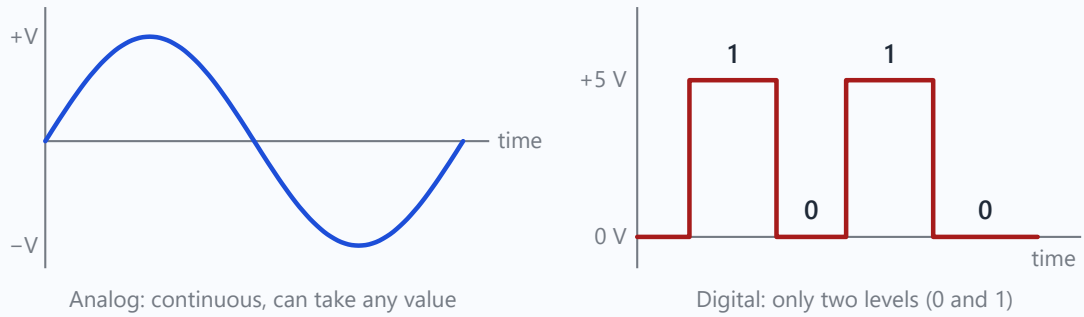


Figure 1.2.4: Analog and digital signals. On the left, an analog sine signal that varies continuously and can take any value; on the right, a digital square wave that switches only between the levels 0 V and 5 V.

Figure 1.2.4 shows why digital representation is preferred. In an analog signal, every voltage value carries different information. So a little noise mixed into the signal changes the information. In a digital signal there are only two levels. In a 5 V system, both 3.9 V and 4.8 V count as "1". Even if noise shifts the signal a little, it is easy to tell which level it is at. The reliability of digital systems comes from this. The logic gates familiar from electronics courses are circuits that work with these two-level signals.

1.2.4.3 The fixed-program computer

Definition 1.2.10

A **fixed-program computer** is a machine designed to solve a specific problem and to run a single predefined program. The program is built into the structure of the machine; to make it do another job, the machine itself has to be changed.

The roots of fixed-program machines go back a long way:

- The abacus and the Antikythera Mechanism (2nd century BC). The Antikythera Mechanism is an ancient Greek device that computed the positions of celestial bodies with gears.
- The Pascaline (Blaise Pascal, 1642) and Leibniz's stepped reckoner (1673).
- Babbage's Difference Engine (1822) and Hollerith's electric tabulating machine (1890). Hollerith's machine was built to count the results of the US census from punched cards.
- The Atanasoff–Berry Computer (ABC) and the Turing Bombe. The Bombe was built in World War II to break the German Enigma cipher.



Figure 1.2.5: The Pascaline. This mechanical machine, driven by gears, can only add and subtract.

The Pascaline in Figure 1.2.5 is a typical example of a fixed-program machine. Its wheels are arranged to do addition; another computation, such as a square root, cannot be done on this machine. Fixed-program devices exist today too: a simple pocket calculator, or the old-style mechanical programmer of a washing machine, works on the same logic. Fixed-program machines laid the foundations of computer science, inspired technological progress and were the forerunners of modern computers.

1.2.4.4 Babbage and the Analytical Engine

Charles Babbage (1791–1871) was an English mathematician and inventor. The Difference Engine was a fixed-program machine designed to compute tables of values of polynomial expressions. Babbage later turned to a far more ambitious design. In 1837 he designed the **Analytical Engine**. This was the first design for a general-purpose, programmable computer.

The Analytical Engine could do mathematical computation and store results. Its real novelty was that it could be programmed for different computations. The parts of today's computer were already in this design:

- the **store**: the memory that holds the numbers;
- the **mill**: the unit that carries out the operations, the counterpart of today's processor;
- input by punched cards and output by a printer.

The punched card idea was borrowed from the weaving loom. In 1804 Joseph Marie Jacquard had developed a loom whose patterns were set by punched cards. When the cards were changed, the pattern changed. Babbage applied the same idea to computation: when the cards were changed, the computation would change. To change what the machine did, it was enough to change the cards, not the machine.

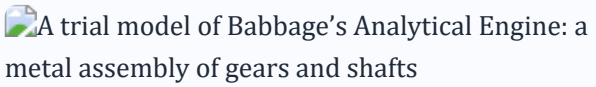


Figure 1.2.6: A trial model of the Analytical Engine (a part of the machine).

The machine could not be completed in Babbage's lifetime. The metalworking of the day was not up to producing thousands of precision parts, and the money could not be found. Today only trial models exist, like the one in Figure 1.2.6. Still, the Analytical Engine is regarded as the forerunner of modern computers, because it was the first to define clearly what parts a general-purpose machine must consist of.

1.2.4.5 Ada Lovelace and the first program

Ada Lovelace (1815–1852) was an English mathematician and writer. In 1843 she translated into English a French article by the Italian engineer Luigi Menabrea about the Analytical Engine, and added her own notes to the translation. The notes were longer than the article itself.

The most famous of the notes is "Note G". In it Lovelace gave a step-by-step method for the machine to compute Bernoulli numbers. The method is a table showing which value is to be written to which memory cell and which operation is to be done in which order. This table is regarded as the first computer program, and Lovelace is counted as the first programmer in the history of computing. Lovelace combined a scientific approach with mathematical thinking in programming.

Lovelace's foresight went beyond the program. She was among the first to notice that the machine could work not only with numbers but with symbols. In her view, if musical notes were also represented by symbols, the machine could compose music. Today's computers processing text, sound and images is the realization of this idea. Lovelace's legacy also gave its name to a programming language: the Ada language, introduced in 1980, bears her name.

This story has a lesson: Lovelace's program was written when the machine did not yet exist. An algorithm is not tied to a machine; it can be designed on paper and followed by hand. The code traces made throughout these notes rest on the same idea.

1.2.4.6 Timeline and the stored program

About a century lies between Babbage and the first electronic computers. Figure 1.2.7 shows the important dates of this period together.

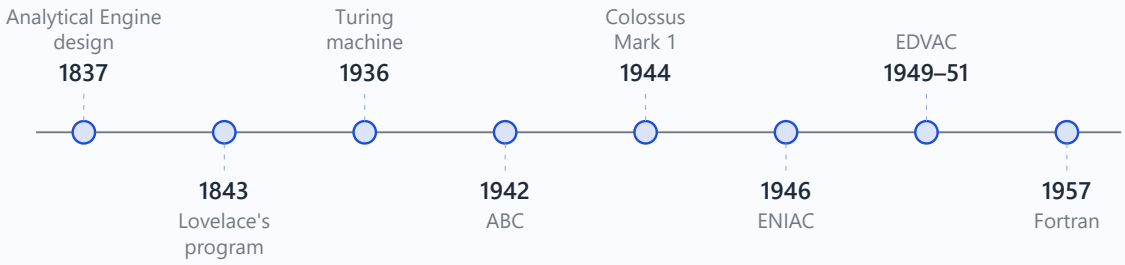


Figure 1.2.7: A short history of the computer: 1837 Analytical Engine design, 1843 Lovelace's program, 1936 Turing machine, 1942 ABC, 1944 Colossus Mark 1, 1946 ENIAC, 1949–51 EDVAC, 1957 Fortran. The intervals on the timeline are drawn evenly; they do not show the real distance between the years.

The timeline divides into three periods:

- **The mechanical period:** Gears and cards. The Pascaline, the Difference Engine and the Analytical Engine belong to this period.
- **The theoretical foundation:** In 1936 Alan Turing defined exactly what the concept of "computable" means and put forward the idea of the universal machine. This work built no machine; it determined what a machine can and cannot do.
- **The electronic period:** The vacuum-tube ABC, Colossus and ENIAC; then the stored-program EDVAC.

The most important step of the electronic period is the **stored-program** idea.

> **Definition 1.2.11**
>
> A **stored-program computer** is a computer that keeps the program in memory just like data. Changing the program means writing a new program into memory; the structure of the machine is left untouched.

The importance of the idea in 1945 lies here: since the program and data sit in the same memory, a program is also data. One program can read, modify or produce another program as input. Compilers do exactly this: `javac` reads a Java source file as input and produces another program as output. The Java virtual machine likewise reads the program in a `.class` file as data and runs it.

> **Definition 1.2.12**
>
> The **von Neumann architecture** is a computer organization in which instructions and data are kept in the same memory. The processor reads instructions from memory in order and executes them. The processor, memory and input/output units communicate over address, data and control paths (the *bus*).

The laptops, phones and servers in use today work basically on this architecture. In a loop, the processor fetches the next instruction from memory, decodes it and executes it; then moves on to the next one. A conditional jump instruction can change this order. The `if` and `while` statements in Java turn into jump instructions of this kind after compilation.

1.2.4.7 The first electronic computers: ABC and Colossus

The **Atanasoff–Berry Computer (ABC)** is the machine developed by John Atanasoff and Clifford Berry between 1937 and 1942 at Iowa State University. It is regarded as the first electronic digital computer. The features of the ABC are as follows:

- Its logic circuits were built with vacuum tubes.

- It processed numbers in the binary (0/1) system.
- Its memory consisted of capacitors on rotating drums. A capacitor being charged or empty represented one bit.
- It could not be programmed. It only solved systems of linear equations; that is, it was a fixed-program machine.

Colossus Mark 1 is the machine that Tommy Flowers and his team built in 1944 at Bletchley Park in England. It is regarded as the first programmable electronic digital computer; it was programmed with switches and plugs. It was used in World War II to break the enemy's encrypted messages. It contained about 1600 vacuum tubes and was far faster and more flexible than mechanical devices. Colossus contributed greatly to the progress of computer technology; however, since it was kept secret for many years after the war, its influence was understood late.

The difference between the two machines sums up the main line of history. The ABC was electronic and binary, but tied to a single job. Colossus was programmable, but changing the program meant changing the switches and plugs by hand. It took a few more years before the program was held in memory.

1.2.4.8 The vacuum tube and the transistor

The common component of the first electronic computers is the **vacuum tube**. A vacuum tube is a glass tube with the air pumped out. It contains three main elements:

- **Cathode:** Emits electrons when heated.
- **Anode** (plate): Collects the electrons the cathode emits.
- **Grid:** Sits between the cathode and the anode. A small voltage applied to the grid switches the electron flow on or off.

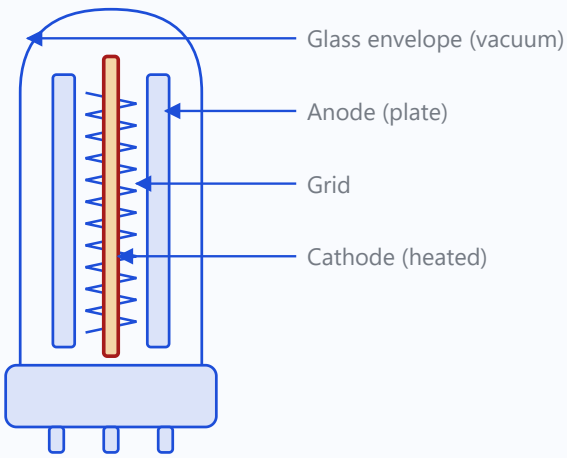


Figure 1.2.8: The structure of a vacuum tube. In the middle of the glass envelope is the heated cathode, around it the grid, and outermost the anode plate.

Being able to switch the current on and off with the grid voltage makes the tube an electronic switch: 1 if current flows, 0 if not. Since this switching involves no moving parts, it is thousands of times faster than mechanical relays. A relay opens and closes within milliseconds, while a tube switches on the order of microseconds.

The drawbacks of the vacuum tube were also large. Tubes get very hot, use a lot of energy and fail often. In a machine with thousands of tubes, a few burning out every day was normal. The solution was the **transistor**, invented in 1947. The transistor does the same switching job far smaller, cooler and more reliably. The MOSFETs studied in electronics courses do this job too. Today's processors contain billions of transistors.

1.2.4.9 ENIAC and EDVAC

ENIAC (1946) is the machine built by John Mauchly and J. Presper Eckert at the University of Pennsylvania (USA). It is regarded as the first large-scale general-purpose electronic computer. It used about 17,500 vacuum tubes instead of mechanical parts, so it was much faster than earlier machines. It could do about 5000 additions per second. It was used for large computations such as artillery firing tables.

ENIAC had two weak points. It held numbers in the decimal system, which complicated its circuits. More importantly, to change the program the cables had to be replugged by hand. Preparing the machine for a new problem could take days.

EDVAC (1949–1951) solved these two problems. Its design was described in a report John von Neumann wrote in 1945; the machine became fully operational in 1951. The innovations of EDVAC are as follows:

- Unlike ENIAC, it used the binary number system.
- It stored the program and the data in the same memory. This is the implementation of the stored-program idea.
- Instructions ran in sequence; conditional jumps were possible.

This design is the von Neumann architecture, the basis of today's computers. What separates ENIAC from EDVAC is not speed but where the program sits. In ENIAC the program was in the cables; in EDVAC it was in memory.

1.2.4.10 The general-purpose computer

Definition 1.2.13

A **general-purpose computer** is a computer that can be programmed according to the user's needs and is designed for a wide range of applications. It consists of a processor, memory and input/output units; the processor processes data according to instructions and produces results.

The general-purpose computer is used in many fields such as business, education and entertainment. It is adapted to different tasks by changing the program: the same laptop plays music a minute ago and compiles Java a minute later. This is its difference from a fixed-program machine. General-purpose computers automate business processes, speed up scientific computation and make communication easier.

Property	Fixed-program	Stored-program
Where the program is	In the structure of the machine (gears, wires, circuits)	In memory, like data
Changing the program	Rebuilding the machine	Loading a new program into memory
What it can do	A single job	Any job that can be programmed
Examples	Pascaline, Difference Engine, ABC	EDVAC, today's computers

Table 1.2.4: Comparison of fixed-program and stored-program computers.

1.2.4.11 The first programs and Fortran

The first programs were mostly for mathematical computations. Uses focused on scientific and military applications: firing tables, nuclear physics calculations, weather forecasting. Programs and data were entered with punched cards. The pattern of holes in each column of a card encoded one character.

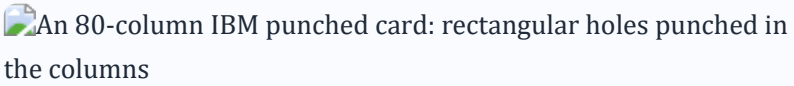


Figure 1.2.9: An 80-column punched card. A card corresponds to roughly one line of a program.

A card like the one in Figure 1.2.9 held 80 characters. A program of a hundred lines was a deck of a hundred cards. If the deck was dropped and the cards got shuffled, the program was ruined. The 80-character line limit seen in some code editors today is a leftover from this period.

The first programs were written in machine language or very close to it. Since every processor has different instructions, a program written for one machine did not run on another. **Fortran** (Formula Translation), the first widely used **high-level programming language**, eased this problem. Fortran was developed at IBM under the leadership of John Backus and released in 1957. It was widely used in scientific computing and is still used in that area today.

A high-level language lets the programmer write expressions close to mathematics such as `K = I + J`. A **compiler** translates this expression into machine instructions. The programmer now thinks in terms of variable names rather than memory addresses. Java (1995) is also a high-level language; the line `int sum = a + b;` continues this tradition.

1.2.5 Turing machines

The word "computation" has a vague meaning in everyday language. What does it mean to say that a problem is "solvable by computer"? With which computer, how much memory, which language? In 1936 Alan Turing defined an extremely simple abstract machine to answer this question independently of all machines. This machine is not fast or practical like a real computer. Its value comes from defining exactly what computation is.

1.2.5.1 The universal Turing machine

> **Definition 1.2.14**
>
> A **universal Turing machine** is an abstract Turing machine that takes as input the description of another Turing machine together with that machine's input, and then simulates that machine.

This model, defined by Alan Turing in 1936, is an abstract model of a general-purpose computer and one of the most fundamental models of computer science. The universal machine can do whatever any Turing machine given to it can do. It can therefore solve every computable problem.

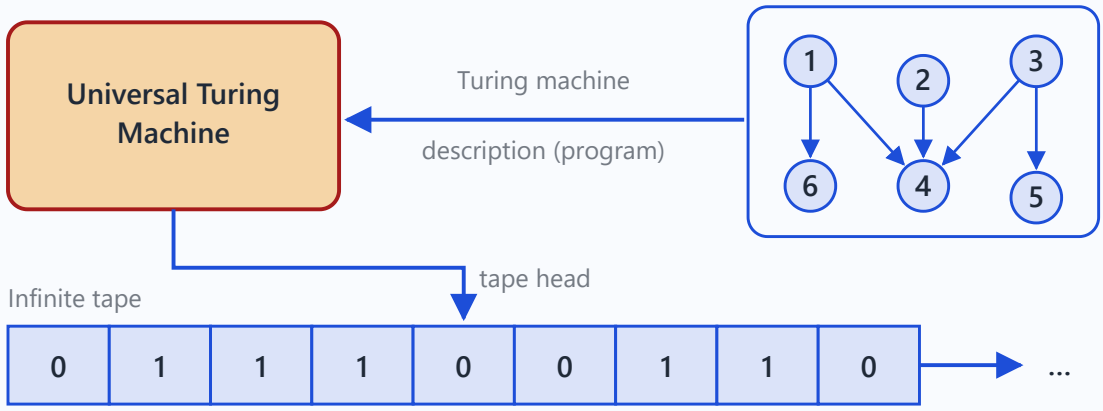


Figure 1.2.10: The universal Turing machine. It takes the description (program) of another Turing machine as input and simulates that machine on its infinite tape.

The universal machine idea is the theoretical counterpart of the stored-program idea. In Figure 1.2.10, the description of the machine to be simulated is read like the other data on the tape. Today's computers work the same way: a single piece of hardware becomes a text editor, a game or a compiler depending on the program in memory. This is the theoretical basis of the universal programmability of modern computers. The stored-program computer and the universal Turing machine are two important turning points of computer technology: one determined how to build the machine, the other what it can do.

1.2.5.2 The parts of a Turing machine

Definition 1.2.15

A **Turing machine** is an abstract model of computation made of three parts:

- **Tape:** A strip of infinite length divided into cells. The input, output and intermediate results are stored on the tape. Each cell holds a symbol (0, 1, blank □ ...).
- **Tape head:** At any moment it points to a single cell. It reads the symbol in that cell, writes a new symbol in its place and moves one cell left or right.
- **State and rules:** The machine's current state (q0, q1, ...) together with the symbol read determines what to write, where the head moves and the next state.

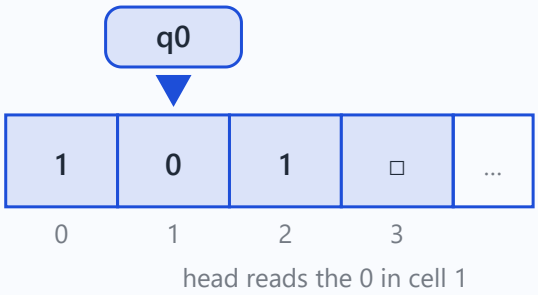


Figure 1.2.11: A snapshot of a Turing machine. The tape holds 1 0 1 □; the head reads the 0 in cell 1; the machine is in state q0.

The rule format is as follows:

> (state, read) → (write, move, new state)

For example, the rule (q0, 0) → (1, right, q0) means: when the machine is in state q0 and there is a 0 under the head, a 1 is written to that cell, the head moves one cell right and the machine stays in state q0. If this rule is applied to the snapshot in Figure 1.2.11, cell 1 becomes 1 and the head moves to cell 2.

Every step of the machine is this simple: read a symbol, write a symbol, move one cell, change the state. All operations such as addition, multiplication and comparison are built from long sequences of these simple steps. Since the tape is infinite, there is no memory limit. This is also its difference from real computers: a real computer's memory is finite.

The components of a Turing machine can be likened to those of a Java program. The tape is the memory where data is kept; the head points at the variable currently being worked on; the rule table is the program; and the state is which line the program is currently at. The analogy is not perfect, but it helps one see that a Turing machine, too, runs a program.

1.2.5.3 Example: a machine that flips bits

One of the simplest Turing machines is the one that turns every 0 on the tape into 1 and every 1 into 0. The machine has only one working state (q0) and three rules. The rules are given in Table 1.2.5.

#	State, read	Write, move, state
1	q0, 0	1, right, q0
2	q0, 1	0, right, q0
3	q0, □	halt

Table 1.2.5: The rule table of the bit-flipping Turing machine.

By the rules, if 0 is read, 1 is written, and if 1 is read, 0 is written, and in both cases the head moves one cell right. When it reaches a blank cell, the machine halts. Let us run the machine with 0110 written on the tape. Initially the state is q0 and the head is at cell 0. Table 1.2.6 shows each step.

Step	Tape (before)	Head	Read	Rule	Tape (after)
1	**0**110□	0	0	1: write 1, right	1110□
2	1**1**10□	1	1	2: write 0, right	1010□
3	10**1**0□	2	1	2: write 0, right	1000□
4	100**0**□	3	0	1: write 1, right	1001□
5	1001□	4	□	3: halt	1001□

Table 1.2.6: The bit-flipping machine running on the tape 0110. The cell the head reads is shown in bold.

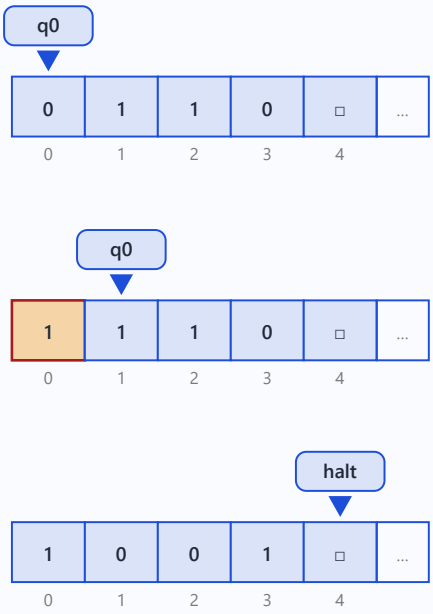


Figure 1.2.12: Three moments from the bit-flipping machine. (a) Start: the tape holds 0110, the head is at cell 0, the state is q0. (b) After step 1: a 1 has been written to cell 0 and the head has moved to cell 1. (c) End: □ was read and the machine halted; the tape holds 1001.

The machine runs four writing steps and one halting step, five steps in total. The tape ends up holding 1001. The number of steps depends on the length of the input: for an input of n bits, the machine runs n writing steps and one halting step, that is, $n + 1$ steps.

This small rule table is the machine's **program**. The universal machine can read such a table from its tape and run it. The same idea can be written in Java with a loop: for each element of an array, if the element is 0 make it 1, if it is 1 make it 0, and move on to the next element. The "move right" step in the Turing machine corresponds to incrementing the index in the loop; halting at a blank cell corresponds to leaving the loop at the end of the array.

Example 1.2.2

Let the same machine run on the tape 1010. The steps: in cell 0, 1 is read and 0 is written (0010); in cell 1, 0 is read and 1 is written (0110); in cell 2, 1 is read and 0 is written (0100); in cell 3, 0 is read and 1 is written (0101); in cell 4, □ is read and the machine halts. There are 4 writing steps and 1 halting step in total, that is, 5 steps. The tape ends up holding 0101.

Common mistake

Forgetting the halting rule. If the table contained only the first and second rules, the machine would not know which rule to apply on reaching a blank cell. In this undefined situation the machine gets stuck; it hangs without producing a correct result. Forgetting the termination condition in a Java loop is a similar mistake: the loop runs past the end of the array and an `ArrayIndexOutOfBoundsException` is thrown.

1.2.5.4 Computability and the halting problem

The Turing machine defines exactly what computation is and what its limits are. The claim that has made this definition accepted is the Church–Turing thesis.

Theorem 1.2.1 (Church–Turing tezi)

Anything that can be computed by a step-by-step method can also be computed by a Turing machine.

Although it is given in a theorem box, the Church–Turing thesis is not a proven mathematical theorem. "A step-by-step method" is an intuitive notion and has no exact mathematical definition. The thesis claims that this intuitive notion is captured exactly by the Turing machine. None of the models of computation proposed so far has turned out to be more powerful than the Turing machine; this is why the thesis is generally accepted.

Definition 1.2.16

A problem is called **computable** if there is an algorithm (equivalently, a Turing machine) that gives the right answer for every input in a finite number of steps. **Computability** is the question of whether a problem can be solved by any algorithm at all.

The really surprising result of Turing's 1936 paper is that not every problem is computable. The most famous example is the **halting problem**: when a given program is run on a given input, does it eventually stop, or does it run forever? Turing proved that there can be no algorithm that answers

this question correctly for every program.



Figure 1.2.13: Among all problems, the computable ones form a subset. The problems in this set are solved by a Turing machine or a Java program. The halting problem lies outside the set; there is no algorithm that solves it.

The idea of the proof rests on constructing a counterexample and can be told in a few sentences. Suppose a method named `halts(p, x)` could be written. Let this method return `true` if program `p` stops when run on input `x` and `false` if it does not, and let it always stop itself. Using this method, a new program, `trouble(p)`, is written. `trouble` asks the question `halts(p, p)`: if the answer is `true` it enters an infinite loop, and if `false` it stops at once. Now let the program `trouble` be run on its own source code. If `trouble(trouble)` stops, then `halts(trouble, trouble)` returns `true` and `trouble` enters an infinite loop; that is, it does not stop. If it does not stop, `halts` returns `false` and `trouble` stops at once; that is, it stops. Both cases lead to a contradiction. The only source of the contradiction is the assumption that the `halts` method can be written; so no such method can be written.

This result has a practical meaning. The Java compiler notices some obvious infinite loops, but a compiler that finds every infinite loop cannot be written. Nor is there a general tool that automatically shows that a program stops on every input. The limit comes not from speed or memory but is a logical one. A faster computer cannot get past this limit. This result is regarded as one of the most important scientific results of the 20th century.

1.2.5.5 Turing completeness and Java

Definition 1.2.17

A programming language or system that can do every computation a Turing machine can do is called **Turing complete**.

Languages such as Java, Python and C are Turing complete. They can do every computation a Turing machine can. For this it is enough for a language to have conditional branching, unbounded repetition (a loop or recursion) and the ability to hold as much data as desired. Languages differ from one another in speed and convenience, not in power. Any computation that can be written in Java can be written in Python as well; only one may be shorter and the other faster.

One consequence of this equivalence is that the choice of language is not a question of "power". The reasons for choosing Java for a project are engineering ones such as type checking, portability and the richness of libraries. The halting problem also holds for all Turing-complete languages: in Java, too, no program can be written that decides whether every program halts.

1.2.6 The limits of computers

The halting problem is the logical limit of computation: no computer can get past it. Besides this there are more everyday limits too. Even a computable problem cannot be solved in practice if the time or memory it needs exceeds what is available. This section covers the limits of physical resources and how these limits show up in a Java program.

1.2.6.1 Resource limits

Computers do complex computations very fast; this power rests on technological progress. Even so, every resource is limited:

- **Processor speed:** Determines how fast computations are done. Today's processors perform billions of operations a second, but this number is finite too. An algorithm whose number of steps grows very fast with the input can take years even on the fastest processor.
- **Memory:** The capacity to store data. More memory means more complex tasks, but memory is limited too. The tape of a Turing machine is infinite; the memory of a real computer is not.
- **Data processing capacity:** Processing large data sets takes time and resources. Even reading millions of records is limited by the speed of the disk.
- **Energy:** Computers consume energy, which is a limited resource. A more powerful computer means more energy. Large data centers can use as much electricity as a small town.

Programmers try to get past these limits by using resources efficiently: they choose algorithms that do the same job in fewer steps and with less memory. The difference between trying every number one by one to find the GCD and using Euclid's algorithm is a good example of this choice.

1.2.6.2 The limits show up in Java too

Since a computer's memory is finite, every number type is also stored with a finite number of bits. This limit sometimes leads to unexpected results. The following program shows three typical cases.

```java
public class Limits {
    public static void main(String[] args) {
        int max = Integer.MAX_VALUE;
        System.out.println(max);
        System.out.println(max + 1);  // overflow
        System.out.println(0.1 + 0.2);    // precision
        System.out.println(5 / 2);        // int division
    }
}
```

The program produces the following output:

```
2147483647
-2147483648
0.30000000000000004
2
```

The four lines need to be explained one by one.

First and second lines: overflow. In Java the `int` type uses 32 bits of memory. Since one of the bits is reserved for the sign, the largest value is $2^{31} - 1 = 2{,}147{,}483{,}647$; this value is held in the constant `Integer.MAX_VALUE`. When 1 is added to this number, the result does not fit in 32 bits. Java raises no error in this case; the result silently wraps around to the smallest value, −2,147,483,648. This event is called **overflow**.

> **Definition 1.2.18**
>
> **Overflow** is when the result of an operation does not fit in the range of values of the type holding the result. In Java, overflow in integer types is silent: the result wraps around to the other end of the range and no error is thrown.

The reason it wraps around is that negative numbers are stored in *two's complement* form. 2,147,483,647 is written in binary as `0111…1` (a 0 and thirty-one 1s). When 1 is added to it, the carry chain turns all the 1s into 0s and the leftmost bit becomes 1: `1000…0`. In two's complement the leftmost bit is the sign bit; this bit pattern denotes $-2^{31} = -2{,}147{,}483{,}648$. The old mechanical odometers of cars behave similarly: after 99,999 they roll over to 00 000.

Overflow leads to insidious bugs in real programs. The mathematical value of the expression `100000 * 100000` is 10^{10}, but Java does this multiplication as `int` and gives the result 1,410,065,408. If one of the operands is made a `long` (`100000L * 100000`), the result comes out right: 10,000,000,000. A `long` is 64 bits and its largest value is about $9.2 \cdot 10^{18}$. If you do not want overflow to pass silently, use methods such as `Math.addExact` and `Math.multiplyExact`; when overflow occurs, these methods throw an `ArithmeticException` with the message "integer overflow".

Third line: precision. The `double` type stores decimal numbers in binary with about 15–16 significant digits of precision. The number 0.1 cannot be written exactly in binary; just as 1/3 goes on forever in decimal as 0.333..., 0.1 is an infinite fraction in binary. The computer cuts this fraction off somewhere and stores the nearest value. When the approximate values of 0.1 and 0.2 are added, the small errors accumulate and the result comes out as 0.30000000000000004. That is why the expression `0.1 + 0.2 == 0.3` gives `false`. Decimal numbers should be compared not with `==` but by checking whether the difference between them is very small: `Math.abs(a - b) < 1e-9`. The loop condition in the Babylonian method is written with this idea.

Fourth line: integer division. The quotient of two `int` values is again an `int`; the fractional part is discarded. The expression `5 / 2` gives 2, not 2.5. The discarding is done toward zero: the expression `-7 / 2` gives −3, not −3.5. For a decimal result at least one of the operands must be a `double`: the expression `5 / 2.0` or `5.0 / 2` gives 2.5.

Division by zero also behaves differently in the two types. `5 / 0` is an integer division and throws `ArithmeticException: / by zero`. `1.0 / 0`, however, gives no error; the result is `Infinity`. The result of the expression `0.0 / 0` is `NaN`, meaning "not a number".

Expression	Result	Reason
`Integer.MAX_VALUE + 1`	`-2147483648`	Overflow: wraps around to the smallest value.
`100000 * 100000`	`1410065408`	Overflow: 10^{10} does not fit in 32 bits.
`100000L * 100000`	`10000000000`	`long` is 64 bits.
`0.1 + 0.2`	`0.30000000000000004`	0.1 cannot be written exactly in binary.
`5 / 2`	`2`	Integer division.
`-7 / 2`	`-3`	Truncated toward zero.
`5 / 2.0`	`2.5`	One operand is a `double`.
`5 / 0`	`ArithmeticException`	Division by zero with integers.
`1.0 / 0`	`Infinity`	Division by zero is not an error for `double`.

Table 1.2.7: Examples of the limits of number types in Java (verified by running with JDK 21).

Common mistake

Thinking that storing the result in a `double` variable makes the division a decimal one. In the line `double avg = 205 / 3;`, `205 / 3` is first computed as an integer and gives 68; then this 68 is converted to 68.0 and stored in the variable. The lost fraction does not come back. The right way is to write `205 / 3.0` or `(double) 205 / 3`.

1.2.6.3 Machines and technology

Technology is a force that affects people's lives; machines are an important part of this development. Machines do jobs fast and efficiently. This power has grown continuously since the industrial revolution. Even so, there are limits to the machines that can be built:

- **Physical limits:** Size, weight and durability are bound by the laws of physics. A transistor cannot be made smaller than a few atoms. A signal cannot travel faster than light; in a 3 GHz processor, light covers only about 10 cm during one clock cycle. The size of the chip therefore limits speed too.
- **Energy limits:** Machines need energy, and energy sources are limited. A more powerful machine needs more energy and more cooling. Most of the electrical energy spent turns into heat; if this heat cannot be carried away, the chip is damaged.
- **Technological limits:** New materials, new design ideas and new algorithms can push the limits. The move from the vacuum tube to the transistor was the overcoming of such a limit.

Understanding limits makes it possible to design better and more efficient machines. In software, the counterpart is algorithms that solve the problem with fewer resources. If hardware cannot go beyond a certain speed, an algorithm that does the same job in fewer steps often gives a bigger gain than a faster machine.

1.2.7 Declarative and imperative knowledge

There are two ways of knowing something. Knowing what a square root is and being able to find the square root of a number are not the same thing. The first is knowing the definition, the second is knowing the method. To get a computer to do a job, the second kind of knowledge is needed. This section separates the two kinds of knowledge and shows both on two classic problems: the square root and the greatest common divisor.

1.2.7.1 Two kinds of knowledge

Knowledge is conscious understanding that helps people or machines make sense of the world and solve problems. It can be expressed in two forms: declarative and imperative. Table 1.2.8 compares the two kinds.

	Declarative knowledge	**Imperative knowledge**
Question	**What** is something?	**How** is something done?
Form	Definition, axiom, fact	Ordered instructions, recipe
Testing it	Verified as true or false	The steps are followed and a result is obtained
Square root	If $y \cdot y = x$, then y is the square root of x	Babylonian method: guess, improve, repeat

	Declarative knowledge	Imperative knowledge
Everyday example	"A cake is a dessert made of flour, sugar and eggs."	Cake recipe: "Beat the eggs, add the flour, bake at 180 °C for 35 min."

Table 1.2.8: Comparison of declarative and imperative knowledge.

A computer needs imperative knowledge: programs write down "how to do it" knowledge. A kitchen apprentice who knows the definition of a cake cannot make a cake; they need the recipe. A computer cannot produce anything starting from a definition either; it must be given steps. Declarative knowledge is not wasted, though: it serves to test whether the result is right. Whether what the recipe produced is really a cake is decided by looking at the definition.

1.2.7.2 Declarative knowledge

Definition 1.2.19

Declarative knowledge is knowledge that is stated explicitly and can be verified as true or false. It says what something is, through axioms or definitions. It also includes stating facts, that is, true and verifiable statements about the world.

"Water boils at 100 °C at sea level" is a fact; it is declarative knowledge. The definition of the square root is declarative too:

> y is the square root of $x \Leftrightarrow y \cdot y = x$ and $y \geq 0$.

With this definition a guess can be tested. Since $4 \cdot 4 = 16$ and $4 \geq 0$, 4 is the square root of 16. Since $5 \cdot 5 = 25 \neq 16$, 5 is not. The condition $y \geq 0$ in the definition is also necessary: although $(-4) \cdot (-4) = 16$, -4 is not counted as the square root of 16.

But the definition does not say how to compute the square root. It is not possible to find $\sqrt{2}$ directly from this definition. By the definition, $\sqrt{2}$ is the number that gives 2 when multiplied by itself; finding out what this number is requires a method. The only thing the definition can do is test a proposed candidate: since $1.4 \cdot 1.4 = 1.96$, 1.4 is not $\sqrt{2}$, but it is close to it.

1.2.7.3 Imperative knowledge: the Babylonian method

Definition 1.2.20

Imperative knowledge is knowledge made of instructions on how to do something. It is given as ordered steps; when the steps are followed, the result is obtained.

An example of imperative knowledge for the square root is the **Babylonian method**. The method finds the square root of a number step by step:

1. Take the value x.
2. Choose an initial guess y_0 (e.g. $y_0 = 1$).
3. If $y_n^2 \approx x$, stop; the answer is y_n.
4. Otherwise compute the new guess: $y_{n+1} = (y_n + x / y_n) / 2$.
5. Go back to step 3.

Why the method works can be explained by a simple observation. If y is larger than the true root, x / y is smaller than it; if y is smaller, x / y is larger. This is because $y \cdot (x / y) = x$, and the two factors cannot both be larger or both smaller than the root. The true root always lies between y and x / y. Their average is the middle of this interval and is closer to the root than either end. With each repetition the guess approaches the root.

Convergence is very fast. If the first guess is close enough to the root, the number of correct digits roughly doubles with each step. The reason is that the error of the new guess is proportional to the square of the old error: if the error is 0.01, it drops to the order of 0.0001 in the next step. Table 1.2.9 shows the first five repetitions for $x = 16$ and $y_0 = 1$.

n	y_n	y_n^2
0	1,000000	1,000000
1	8,500000	72,250000
2	5,191176	26,948313
3	4,136665	17,111995
4	4,002258	16,018065
5	4,000001	16,000005

Table 1.2.9: Computing $\sqrt{16}$ with the Babylonian method ($y_0 = 1$). Values are rounded to six digits.

The first rows of the table show a property of the method. The initial guess 1 is far below the true root (4); so $x / y = 16$ is very large and the first average is 8.5. The guess jumps above the root at once. From then on all the guesses stay above the root and approach it at each step. Convergence is slow in the first steps and speeds up as it nears the root: at step 3 the error is about 0.14, at step 4 0.002, and at step 5 0.0000006. Five repetitions find a value very close to 4.

Example 1.2.3

Let $\sqrt{2}$ be computed taking $y_0 = 1$.

$y_1 = (1 + 2/1) / 2 = 1{,}5$; $y_1^2 = 2{,}25$.
$y_2 = (1{,}5 + 2/1{,}5) / 2 = (1{,}5 + 1{,}333333) / 2 \approx 1{,}416667$; $y_2^2 \approx 2{,}006944$.
$y_3 = (1{,}416667 + 2/1{,}416667) / 2 \approx 1{,}414216$; $y_3^2 \approx 2{,}000006$.

The true value is $\sqrt{2} \approx 1.414214$. In three repetitions five decimal digits have been found correctly. What the definition could not do for $\sqrt{2}$, the method does in a few operations.

1.2.7.4 The Babylonian method in Java

The steps of the Babylonian method can be translated directly into Java. The step "if $y^2 \approx x$, stop" becomes a `while` loop that continues as long as the difference $|y \cdot y - x|$ is larger than a small threshold (here 0.001).

```
double x = 16;   // find its square root
double y = 1;    // initial guess y0
while (Math.abs(y * y - x) > 0.001) {
    y = (y + x / y) / 2;   // new guess
    System.out.println(y);
}
System.out.println("Square root ≈ " + y);
```

The lines of the code correspond to the steps of the algorithm as follows:

- `double x = 16;` is step 1. `x` is a floating-point variable; Java automatically converts the integer 16 to 16.0.
- `double y = 1;` is step 2: the initial guess.
- `while (Math.abs(y * y - x) > 0.001)` is the inverse of step 3. Instead of "stop if approximately equal", it says "continue if not approximately equal". `Math.abs` gives the absolute value; it is needed because the difference may be negative.
- `y = (y + x / y) / 2;` is step 4. The old value of `y` is used on the right side and the new one is written to the same variable.
- At the end of the loop, control returns to the `while` line; this is step 5.

Iteration	$\lvert y \cdot y - x \rvert$ in the condition	Condition	New `y` (printed)
1	15	true	`8.5`
2	56,25	true	`5.1911764705882355`
3	10,9483	true	`4.136664722546242`
4	1,112	true	`4.002257524798522`
5	0,0180653	true	`4.000000636692939`
–	$\approx 5{,}1 \cdot 10^{-6}$	false	The loop ends; `Square root ≈ 4.000000636692939` is printed.

Table 1.2.10: Trace of the Babylonian method program (`x` = 16.0). The printed values are the exact values Java gives.

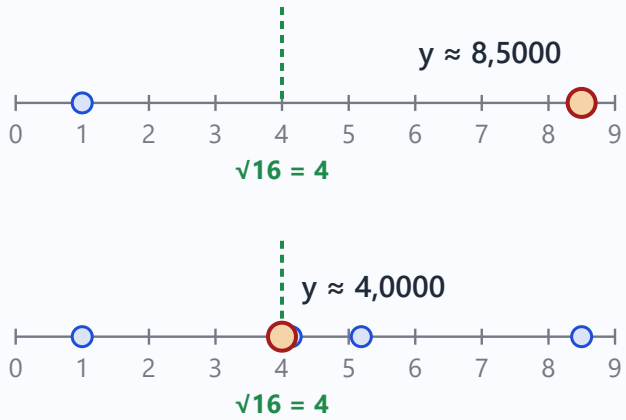


Figure 1.2.14: The guesses of the Babylonian method on the number line. (a) After the first guess 1, the new guess jumps to 8.5. (b) Final state: the guesses 1, 8.5, 5.191, 4.137, 4.002 and 4.000 approach $\sqrt{16} = 4$ in turn.

The program prints not exactly 4 but an approximation very close to 4. There are two reasons. First, the loop stops when the difference falls below 0.001; exact equality is not awaited. Second, `double` has finite precision anyway. If a smaller threshold is chosen (e.g. `1e-12`), the result gets even closer and the loop runs just one or two more iterations.

Common mistake

Writing the loop condition as `while (y * y != x)`. `double` values often never reach exact equality; since a root like $\sqrt{2}$ cannot be written exactly in binary, `y * y` may never be exactly 2.0. In that case the condition is never false and the loop runs forever. With floating-point comparisons, always use a threshold.

1.2.7.5 GCD: two definitions

Definition 1.2.21

The **greatest common divisor** (GCD) of two integers is the largest positive integer that divides both exactly.

For example, the divisors of 12 are 1, 2, 3, 4, 6, 12; the divisors of 18 are 1, 2, 3, 6, 9, 18. The common divisors are 1, 2, 3 and 6; the largest is 6, so GCD(12, 18) = 6.

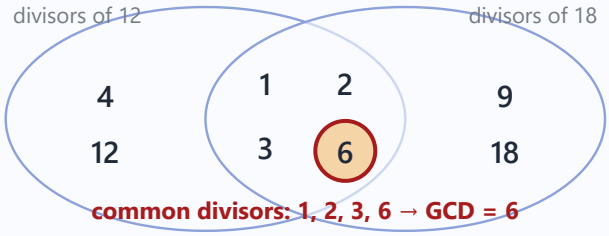


Figure 1.2.15: The divisors of 12 and of 18. The intersection is 1, 2, 3, 6; the largest of these, 6, is the GCD.

The GCD also has two definitions. The **declarative definition** says what the GCD is. d is a common divisor of a and b if, for some integers x and y, $a = d \cdot x$ and $b = d \cdot y$. It is also the largest of the numbers satisfying this condition. This definition serves to test a candidate: 6 divides both 12 and 18, and there is no larger common divisor. But the definition does not give a way to find the GCD. Listing all the divisors, as in Figure 1.2.15, works for small numbers and is very slow for large ones.

The **imperative definition** says how to find the GCD: Euclid's algorithm. It repeats division and taking the remainder until the remainder is 0. Appearing in Euclid's *Elements* around 300 BC, this method is one of the oldest known algorithms.

1.2.7.6 Euclid's algorithm

The steps of the algorithm are as follows:

1. Take two positive integers a and b ($a \geq b$).
2. Divide a by b and call the remainder r: $r = a \bmod b$.
3. If $r = 0$, stop: GCD = b.
4. Otherwise assign b to a and r to b.
5. Go back to step 2.

The correctness of the algorithm rests on a single observation.

Theorem 1.2.2 (Euclid)

If a and b are positive integers and $r = a \bmod b$, then $\text{GCD}(a, b) = \text{GCD}(b, r)$.

Proof. By the division rule, for some integer k, $a = k \cdot b + r$, that is, $r = a - k \cdot b$. Every d dividing a and b also divides the difference $a - k \cdot b$, that is, r. Conversely, every d dividing b and r also divides the sum $k \cdot b + r$, that is, a. So the common divisors of the pair (a, b) and the common divisors of the pair (b, r) form the same set. The largest element of the same set is also the same. ■

The theorem says what the algorithm preserves at each step: the numbers get smaller but the GCD does not change. Since the remainder is always smaller than the divisor ($r < b$), the numbers strictly decrease at each step and eventually the remainder becomes 0. When the remainder is 0, b divides a exactly; in this case GCD(a, b) = b. By the theorem, this result is also the GCD of the original numbers.

a	b	$r = a \bmod b$	Division
48	18	12	48 = 2 · 18 + 12
18	12	6	18 = 1 · 12 + 6
12	**6**	0	12 = 2 · 6 + 0

Table 1.2.11: GCD(48, 18) by Euclid's algorithm.

After three divisions the remainder is 0 and GCD(48, 18) = 6 is found. The same result by listing divisors would be found by writing out the ten divisors of 48 and the six of 18. The difference becomes marked with large numbers. In Euclid's algorithm the first number is halved at least every two steps. If $b \le a/2$, the remainder is smaller than b, hence than $a/2$. If $b > a/2$, the remainder is $a - b$, which is also smaller than $a/2$. This is why even for numbers in the billions the number of steps does not exceed a few dozen.

Example 1.2.4

Find GCD(1071, 462).

1071 = 2 · 462 + 147, so r = 147. (a, b) becomes (462, 147).
462 = 3 · 147 + 21, so r = 21. (a, b) becomes (147, 21).
147 = 7 · 21 + 0, so r = 0.

The algorithm stops and GCD = 21 is found. Three divisions were performed. Check: 1071 = 21 · 51 and 462 = 21 · 22; 51 and 22 have no common divisor larger than 1.

1.2.7.7 Euclid's algorithm in Java

```
int a = 48, b = 18;   // a >= b
int r = a % b;        // remainder
while (r != 0) {
    a = b;
    b = r;
    r = a % b;
}
System.out.println("GCD = " + b);
```

The `%` operator in Java gives the remainder; it corresponds to the "mod" operation in the algorithm. The code follows the steps of the algorithm as follows:

- The first two lines are steps 1 and 2: the numbers are taken and the first remainder is computed.
- `while (r != 0)` is the inverse of step 3: continue if the remainder is not 0.
- The first two lines in the loop body are step 4. The order matters: first `a = b`, then `b = r`. If they were written in the opposite order, the old value of `b` would be lost.
- The line `r = a % b;` is the repetition of step 2; control returns to the `while` line.
- When the loop ends, the last divisor is in `b` and is printed as the result.

Step	Statement	`a`	`b`	`r`
1	`int a = 48, b = 18;`	**48**	**18**	–
2	`r = a % b;`	48	18	**12**
3	`r != 0` → true	48	18	12
4	`a = b;`	**18**	18	12
5	`b = r;`	18	**12**	12
6	`r = a % b;`	18	12	**6**
7	`r != 0` → true	18	12	6
8	`a = b;`	**12**	12	6
9	`b = r;`	12	**6**	6
10	`r = a % b;`	12	6	**0**
11	`r != 0` → false	12	6	0
12	`println`	12	6	0

Table 1.2.12: Trace of the Euclid program (`a` = 48, `b` = 18). The changed cell is shown in bold.

The program prints "GCD = 6". In the trace, at steps 4 and 8, `a` and `b` briefly hold the same value (18 and 18, 12 and 12). This is not a bug; it is an intermediate state because the assignments are made in order, and `b` is updated on the next line.

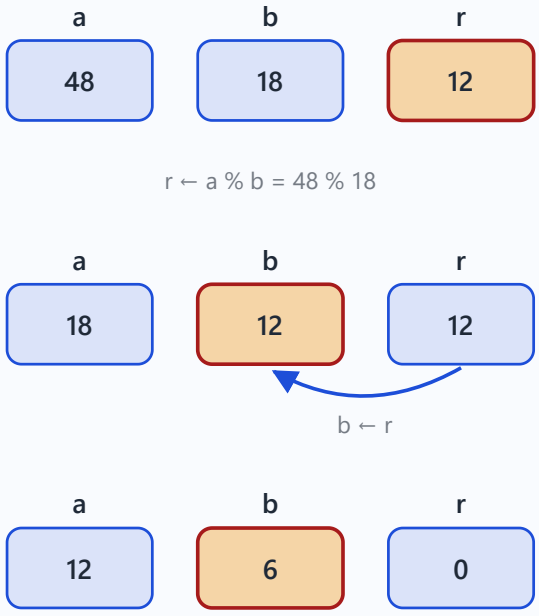


Figure 1.2.16: Three moments from the trace of the Euclid program. (a) The first remainder: 48 % 18 = 12. (b) After the shift, `a` = 18, `b` = 12. (c) End: `r` = 0, the loop ends; the last divisor `b` = 6 is the result.

The code assumes $a \geq b$, but it works correctly even if this condition does not hold. If it starts with `a = 18, b = 48`, the first remainder is 18 % 48 = 18. The first iteration merely swaps the two numbers (`a` = 48, `b` = 18), and the algorithm finds the same result with one extra division: four divisions, GCD = 6.

Common mistake

Printing `a` as the result. When the loop ends, `a` = 12, and 12 is not the GCD. The GCD is the divisor that makes the remainder 0; that is, at the end of the loop it sits in `b`. Another mistake is forgetting that `b` can be 0: if it starts with `b = 0`, the `a % b` operation in the first line throws `ArithmeticException: / by zero`.

1.2.8 Algorithm and problem

The Babylonian method and Euclid's algorithm are two examples of imperative knowledge. What they have in common is that they solve a problem with exact steps. Such sequences of steps are called algorithms. The concept of the algorithm is at the center of computer science; programming is largely expressing algorithms in a language.

1.2.8.1 Algorithm

Definition 1.2.22

An **algorithm** is a method for solving a problem step by step; it is a recipe that fixes the steps to be followed. An algorithm:

- specifies the steps clearly,
- puts the steps in a logical order,
- finishes in a finite number of steps,
- aims at solving the problem.

An algorithm determines how a task is to be done and helps simplify complex problems. It is the basis of computer programs: **programming** is the process of expressing algorithms in a programming language. Computer science too largely consists of the design and analysis of algorithms. A good programmer is one who can choose and implement the right algorithm for a problem.

The four properties in the definition each matter.

- **Clarity:** Each step must be understood in only one way. "Make the number a bit larger" is not a clear step; "add 1 to the number" is. Since a computer does not interpret, it cannot execute a step that is not clear.
- **Order:** The order of the steps changes the result. In Euclid's algorithm, if the order of the assignments `a = b` and `b = r` is changed, the algorithm gives a wrong result. In a cake recipe too, the flour is added before baking.
- **Finiteness:** An algorithm must eventually stop for every input. The instruction "add 1 and print; repeat this without stopping" is clear and ordered, but since it never ends it is not an algorithm. That the Babylonian method stops is guaranteed by the difference threshold, and that Euclid's algorithm stops by the remainder shrinking at every step.
- **Purpose:** An algorithm is written to solve a specific problem and must give the right result.

A well-designed algorithm must be both **correct** and **efficient**. Correctness means giving the right result for every valid input. Efficiency means finding that result in a reasonable time and with a reasonable amount of memory. For the GCD, listing the divisors is correct but inefficient; Euclid's algorithm is both correct and efficient.

The word "algorithm" derives from the name of the mathematician al-Khwarizmi, who lived in 9th-century Baghdad. In the Latin translation of his book on calculating with Hindu-Arabic numerals, his name was written "Algoritmi". Over time the word came to mean a calculation method with fixed rules.

Algorithms are often written not in a programming language but in **pseudocode**. Pseudocode is a way of describing an algorithm in semi-natural language without being tied to a specific language. It does not bother with details such as semicolons and type declarations; it concentrates on the logic of the steps. Patterns used in these notes such as `while … do`, `set … to` and `repeat … until` are common forms of pseudocode.

1.2.8.2 Input, process, output

Before an algorithm is written, the problem itself must be defined clearly. What the program or algorithm is supposed to do is determined; the necessary steps and the output are laid out. Problem specification has two components:

- **Input:** The information or data needed to solve the problem.
- **Output:** The result of the problem or the information asked for.

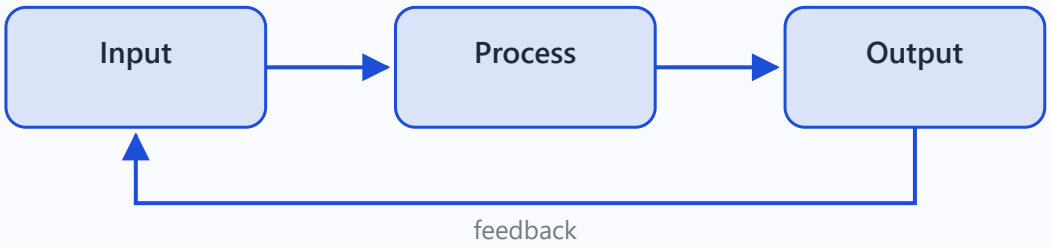


Figure 1.2.17: The input, process and output boxes are connected by arrows from left to right. The arrow going from output back to input shows feedback.

The layout in Figure 1.2.17 is the three stages of computation applied to problem definition. Once the input and output are fixed, the process is the algorithm that connects the two. For Euclid's algorithm, the input is two positive integers and the output is their GCD. For the Babylonian method, the input is a positive number and an initial guess, and the output is an approximate value of the square root.

The return arrow in the figure shows **feedback**: correcting the input or the process by looking at the output. A thermostat is a good example. It measures the temperature (input), compares it with the desired value (process), and switches the heater on or off (output). Switching the heater on changes the temperature of the room, that is, the next input. Closed-loop control systems in electrical and electronics engineering work with this arrangement too. The Babylonian method is also a kind of feedback: the square of each guess is compared with x, and the result determines the next guess.

1.2.8.3 Clarifying a vague request

Six questions help when defining a problem: *What? Who? Where? When? How? Why?* These questions are especially useful for clarifying vague requests given in everyday language.

The request "compute the grade average" is vague. Before writing a program, the answers to at least three questions must be known:

- How many grades are there? Is the number fixed (three grades), or will the user enter it?
- Do the grades have equal weights? If the midterm is 40% and the final 60%, a simple average gives a wrong result.
- Will the result be an integer or a decimal? Will rounding be applied?

A program written without answering these questions may solve the wrong problem. The program compiles without errors and runs without errors, but does not do what was wanted. Finding such mistakes is much harder than finding compile errors, because the compiler gives no warning.

Example 1.2.5

Write the input, output and pseudocode of a program that finds the average of three grades. Let the grades be 40, 75 and 90.

Input: three integers (`n1` , `n2` , `n3`). **Output:** a decimal number, the average. **Pseudocode:** read the three grades → divide the sum by 3.0 → write the result.

The sum is 40 + 75 + 90 = 205. In Java the expression `205 / 3.0` gives `68.33333333333333` . Had the divisor been written as 3, integer division would be performed and the result would be 68. Specifying the output as "decimal" makes it possible to set up this detail correctly from the start.

Example 1.2.6

Let the course grade be computed as "midterm × 0.4 + final × 0.6". The input is two integers (midterm and final) and the output is a decimal number. For a midterm of 50 and a final of 80, the average is 50 · 0.4 + 80 · 0.6 = 20 + 48 = 68. In Java the expression `50 * 0.4 + 80 * 0.6` gives `68.0` . For the same two grades the simple average would be 65; not asking about the weights would have led to an error of 3 points.

1.2.9 Algorithmic operations

The Babylonian method, Euclid's algorithm and the grade average solve different problems, but all are built from the same three kinds of operations. These three kinds exist in every programming language and form the basic structure of algorithms.

1.2.9.1 Three kinds of operations

- **Sequential operation:** Each operation runs after the previous one, in order. For example: read the data, process it, produce the result.
- **Conditional operation:** If a condition holds one operation is done, if not another. For example: is the number positive?
- **Iterative operation:** An operation is repeated as long as a condition holds. For example: summing all the numbers in a list.

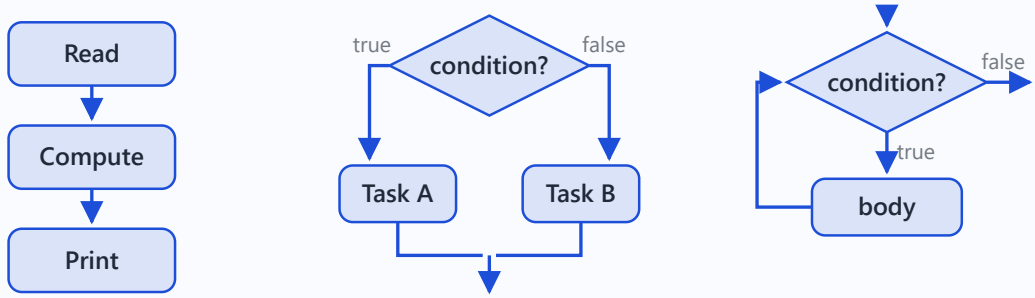


Figure 1.2.18: Flowcharts of the three kinds of operations. (a) Sequential: the Read, Compute, Print boxes one below the other. (b) Conditional: if the condition is true Task A is done, if false Task B; then the paths merge. (c) Iterative: while the condition is true the body runs and control returns to the condition; when the condition is false the loop is left.

All algorithms can be built by nesting the three structures in Figure 1.2.18 inside one another. The body of Euclid's algorithm is three sequential assignments; this body is inside a loop; a condition decides whether the loop continues. In Java, sequential operation is written by writing statements one under the other, conditional operation with `if` / `else`, and iterative operation with `while`, `do-while` and `for`.

It is also important that these three kinds help keep complexity under control. If the flow of a program is built only from these three structures, every part has a single entry and a single exit. The program can be read and traced from top to bottom. This idea became established in the 1960s and 1970s under the name structured programming and influenced all of today's languages, Java included.

1.2.9.2 while: condition first

An iterative operation repeats certain actions until the continuation condition becomes false. It is useful for walking through lists and processing data. A loop has two parts: the **continuation condition** and the **body**. Whether the condition is tested before or after the body gives rise to two different kinds of loop.

In the first kind the condition is tested first. In pseudocode this structure is written with `while … do`. The following example increases *s* by 1 and decreases *i* by 1 as long as $i > 0$.

```
while i > 0 do
    set s to s + 1
    set i to i - 1
```

The Java counterpart is almost one to one:

```
int s = 0, i = 3;
while (i > 0) {
    s = s + 1;
    i = i - 1;
}
```

Iteration	Condition `i > 0`	`s` (end of iteration)	`i` (end of iteration)
Start	–	0	3
1	3 > 0 true	1	2
2	2 > 0 true	2	1
3	1 > 0 true	3	0
–	0 > 0 false	3	0

Table 1.2.13: Trace of the `while` loop (`s` = 0, `i` = 3).

The loop runs three times and ends with `s` = 3, `i` = 0. The condition is tested four times: true in three of them, false in the last. In general, if `i` starts at *n*, the loop runs *n* times and `s` ends up as *n*. Since the condition is tested first, if `i` is 0 to begin with the body never runs.

1.2.9.3 repeat … until and do … while

In the second kind the body runs first and the condition is tested afterwards. In pseudocode this structure is written with `repeat … until`. The following example prints *k* and increases it by 1 until the value of *k* is greater than *n*.

```
repeat
    print k
    set k to k + 1
until k > n
```

Java has no `repeat … until`. Its counterpart is `do … while`, but the condition is **inverted**. `until` stops when the condition becomes true; `while` continues as long as the condition is true. "Until $k > n$" means the same as "as long as $k \le n$".

```java
int k = 1, n = 3;
do {
    System.out.println(k);
    k = k + 1;
} while (k <= n);
```

For $k = 1$, $n = 3$ the program prints 1, 2 and 3, one under the other. The body runs three times; at the end of the third iteration $k = 4$, `4 <= 3` is false and the loop ends.

	Condition first	Body first
Pseudocode	`while condition do …`	`repeat … until condition`
Java	`while (condition) { … }`	`do { … } while (condition);`
Meaning of the condition	Continue while true	In pseudocode: stop when true. In Java: continue while true.
The body runs at least	0 times	1 time

Table 1.2.14: Comparison of loops that test the condition first and the body first.

The difference between the two structures shows when the condition is false from the start. If it starts with `k = 5, n = 3`, the `do … while` loop prints 5 once, then tests the condition and stops. A `while (k <= n)` loop written with the same condition, on the other hand, never runs the body and prints nothing. In cases where the body must run at least once, `do … while` is a natural choice. Asking the user for input until they enter a valid value is a typical example: the value must be read at least once.

Common mistake

Carrying the `until` condition of pseudocode over to Java unchanged. If `while (k > n);` is written in place of `until k > n`, the meaning is reversed. For $k = 1$, $n = 3$ the body runs once and 1 is printed. Then `2 > 3` is false and the loop ends immediately. The expected output is 1, 2, 3, but only 1 is printed. When an `until` condition is carried over to Java, it is always negated.

1.2.9.4 The infinite loop

Definition 1.2.23

An **infinite loop** is a loop that never ends because its continuation condition never becomes false.

For a loop to end, the body must change the variable that affects the condition, and this change must eventually make the condition false. In the `while` example, if the line `i = i - 1;` is forgotten, `i` stays 3 forever, `3 > 0` is always true and the loop never ends. Similarly, if `i = i + 1;` is written, `i` grows and the condition again stays true all the time. In this second case `i` eventually reaches the limit of `int`, overflows and becomes negative; the loop may end, but after a very long time and with a wrong result.

The compiler cannot, in general, catch infinite loops. This is also a consequence of the halting problem: there is no general method that automatically shows that a loop will end for every input. A stuck program is stopped with the stop button in the development environment (IDE) or with Ctrl+C in the console.

Asking three questions when writing a loop largely prevents infinite loops. Does the variable in the condition get a correct value before the loop? Does the body change this variable? Does the change move in the direction that will eventually make the condition false? If the answers to all three questions are "yes", the loop ends.

1.2.10 Common mistakes

When moving from pseudocode to Java, some mistakes are made very often. Some of them the compiler catches and reports an error message; others compile but run wrongly. The messages below were obtained with the `javac` compiler of JDK 21.

Common mistake

Forgetting the semicolon. The line `int a = 5` gets the message `error: ';' expected` from the compiler. In pseudocode line endings are free; in Java every statement ends with `;`. The compiler marks the place where the semicolon is expected with a caret at the end of the line.

Common mistake

Confusing `=` with `==`. The line `if (x = 5)` gives the message `error: incompatible types: int cannot be converted to boolean`. `=` is the assignment operator, and the value of the expression `x = 5` is 5, that is, an `int`. `if` expects a `boolean` condition. For comparison `==` is used.

Common mistake

Not giving an initial value. The lines `int total; System.out.println(total);` give the message `error: variable total might not have been initialized`. Java requires local variables to get a value before they are used. A variable that accumulates a sum should be initialized with `int total = 0;`.

Common mistake

Integer division. The expression `5 / 2` gives 2 and `9 / 5` gives 1. For a decimal result write `5 / 2.0` or `5.0 / 2`. In converting Celsius to Fahrenheit, the line `int f = 9 / 5 * c + 32;` gives 57 instead of 77 for `c` = 25, because `9 / 5` is 1. Moving the multiplication first (`c * 9 / 5 + 32`) gives 77 in this example, but for 37 °C it finds 98 instead of 98.6; the right solution is `c * 9.0 / 5 + 32`.

Common mistake

The decimal separator. On a computer with Turkish settings, the `nextDouble()` method of the `Scanner` class expects a comma as the decimal separator: if "1,75" is entered, 1.75 is read. If "1.75" is typed, an `InputMismatchException` is thrown. In Java source code, however, the decimal separator is always a point (`1.75`); these two rules must not be confused.

Common mistake

Loop logic errors. Initializing the `maxVal` variable with 0 when finding the maximum gives a wrong result for a list whose elements are all negative: for the list {−4, −2, −7} the result is 0 instead of −2. The right way is to start with the first element of the list. Forgetting to update the condition variable also leads to an infinite loop. The compiler catches neither of these mistakes; the program compiles and runs, but gives a wrong result or never ends.

1.2.11 Summary

- **Programming** is writing the instructions that tell a computer step by step what to do. **Computation** is the process of processing data with mathematics and logic to produce a result. Computation is the work being done; programming is the recipe for that work.
- **Computer science** rests on logic, problem solving and creativity; technology is only a tool in this science. The programs we write affect real people, and this brings a responsibility.

- **Computational thinking** has four skills. Decomposition splits the problem into subproblems, and pattern recognition sees the repeating work. Abstraction discards unnecessary detail, and algorithm design writes the solution in finite, clear steps.
- In a **fixed-program** machine the program is built into the hardware. In a **stored-program** machine the program sits in memory like data and can be changed. EDVAC and the von Neumann architecture are the basis of today's computers.
- The main line of history: Babbage's Analytical Engine (1837) and Lovelace's program (1843); Turing's theoretical model (1936); the vacuum-tube ABC, Colossus and ENIAC; the stored-program EDVAC; the transistor (1947) and Fortran (1957).
- A **Turing machine** consists of a tape, a tape head and a rule table. The universal Turing machine reads the description of another machine and simulates it. By the Church–Turing thesis, anything computable by a step-by-step method can be computed by a Turing machine.
- Not every problem is computable: there is no algorithm that solves the **halting problem** for every program. Turing-complete languages such as Java are equal in power; they differ in speed and convenience.
- Resources are limited, and these limits show up in Java too: `int` overflows, `double` holds approximate values, and the quotient of two `int`s is an `int`.
- **Declarative knowledge** says what something is and serves for testing. **Imperative knowledge** says how to do it, and that is what a computer needs. The Babylonian method finds the square root and Euclid's algorithm the GCD, both imperatively.
- An **algorithm** consists of clear, ordered, finite and purposeful steps. Before an algorithm is written, the **input** and **output** of the problem are determined; vague requests are clarified with questions.
- Every algorithm is built from **sequential**, **conditional** and **iterative** operations. `while` tests the condition first, `do … while` afterwards. The `repeat … until` of pseudocode is written in Java as `do … while` with the condition negated. If the body does not eventually make the condition false, the loop goes on forever.

1.2.12 Key terms

computation computational thinking decomposition abstraction algorithm pseudocode bit byte stored-program computer von Neumann architecture compiler Turing machine computability overflow floating-point number integer division declarative / imperative knowledge greatest common divisor (GCD) Euclidean algorithm input / output sequential flow conditional flow while loop do-while loop infinite loop

1.2.13 Test Yourself

1. A program is to be written that finds the busiest hour of the day in a parking lot. Counters at the entry and exit gates record the time of every passing vehicle. Which skill of computational thinking does each of the following decisions correspond to? (a) The license plates and colors of the vehicles are ignored. (b) The problem is split into "read the records, compute the number of vehicles inside for each hour, find the maximum, write it". (c) It is noticed that the job of "finding the maximum" resembles the loop in the grade average.

   Answer

   (a) Abstraction: only the times of passing and the direction (entry or exit) are needed for the result. (b) Decomposition: the big problem has been split into four subproblems. (c) Pattern recognition: the same step repeated for each record can be written with a loop. After these three steps, algorithm design writes the subproblems as ordered steps.

2. What is the basic difference between ENIAC and EDVAC? Which one is a stored-program computer?

   Answer

   In ENIAC the program was set up with cable connections; for a new job the cables were replugged by hand. It also held numbers in the decimal system. EDVAC stored the program and data in the same memory and used the binary system. The stored-program computer is EDVAC; its design is the basis of the von Neumann architecture.

3. How many steps does the bit-flipping Turing machine run on the tape 111000? What is left on the tape when the machine halts?

   Answer

   There is one writing step for each cell: 6 writing steps. At the seventh step □ is read and the machine halts. It runs 7 steps in total; the tape holds 000111. In general an n-bit tape needs $n + 1$ steps.

4. Compute $\sqrt{9}$ with the Babylonian method in three steps, taking $y_0 = 1$. Which side of the root do the guesses stay on?

   Answer

   $y_1 = (1 + 9) / 2 = 5$; $y_2 = (5 + 9/5) / 2 = 3.4$; $y_3 = (3.4 + 9/3.4) / 2 \approx 3.023529$. The initial guess is below the root; at the first step the guess goes above the root and then always stays above, approaching 3.

5. Find GCD(252, 105) with Euclid's algorithm. How many divisions are performed? How is the result checked?

   Answer

   252 = 2 · 105 + 42; 105 = 2 · 42 + 21; 42 = 2 · 21 + 0. After three divisions the remainder is 0 and GCD = 21 is found. Check: 252 = 21 · 12 and 105 = 21 · 5; 12 and 5 have no common divisor larger than 1.

6. What do the following two lines print?

   ```
   System.out.println(Integer.MAX_VALUE + 1 == Integer.MIN_VALUE);
   System.out.println(7 / 2 * 2.0);
   ```

   Answer

   The first line prints `true`: adding 1 to the largest `int` value overflows and the result wraps around to the smallest value. The second line prints `6.0`: operations are done from left to right, `7 / 2` first becomes 3 by integer division, then 3 · 2.0 = 6.0.

7. Two loops starting with `k` = 0 are given. What is `k` at the end of each?

   ```
   do { k = k - 3; } while (k > 0);
   while (k > 0) { k = k - 3; }
   ```

   Answer

   `do … while` runs the body once before the condition: `k` becomes −3, `-3 > 0` is false and the loop ends; the result is −3. `while` tests the condition first: since `0 > 0` is false, the body never runs; the result is 0.

8. In the Average program, if the division line were written as `double avg = total / grades.length;`, that is, without the `(double)` cast, what would be printed for the grades 40, 75 and 90? Why?

   Answer

   "Average: 68.0" is printed. `total` (205) and `grades.length` (3) are both `int`s; the division is done as integer division and gives 68. This value is converted to `double` only at the assignment; the lost fraction does not come back.

9. The sentence "A number is prime if it is an integer greater than 1 with no positive divisors other than 1 and itself" is what kind of knowledge? Give an example of the other kind on the same

subject.

Answer

It is declarative: it defines what a prime number is and serves to test a candidate. An imperative counterpart could be: "Let $n > 1$. For each d from 2 up to $\sqrt{n}$, look at $n \bmod d$; if a d with remainder 0 is found, write 'not prime', otherwise write 'prime'." This recipe says step by step how the decision is made.

1.2.14 Further reading

- J. M. Wing, "Computational Thinking", *Communications of the ACM* 49(3), 2006: the three-page article that popularized the term computational thinking. It discusses why this way of thinking is needed by everyone.
- J. V. Guttag, *Introduction to Computation and Programming Using Python* (3rd ed., 2021), Chapter 1: presents the distinction between declarative and imperative knowledge and the guess-and-improve method for the square root, using the same examples as these notes.
- C. Petzold, *Code: The Hidden Language of Computer Hardware and Software* (2nd ed., 2022): explains in non-technical language how a computer is built up piece by piece starting from relays and switches, binary numbers and the stored-program idea.
- R. Sedgewick and K. Wayne, *Computer Science: An Interdisciplinary Approach* (2016), the sections on Turing machines and computability: gives Turing machine examples, the universal machine and the proof of the halting problem in connection with Java programs.

1.2.15 References

1. H. M. Deitel, P. J. Deitel, *Java: How to Program*, 11th ed., Pearson, 2017.
2. A. B. Downey, C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019.
3. J. V. Guttag, *Introduction to Computation and Programming Using Python*, 3rd ed., MIT Press, 2021.
4. Oracle, *The Java Tutorials*: “Language Basics”, docs.oracle.com/javase/tutorial.
5. C. Petzold, *Code: The Hidden Language of Computer Hardware and Software*, 2nd ed., 2022.
6. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016.
7. A. M. Turing, “On Computable Numbers, with an Application to the Entscheidungsproblem”, *Proceedings of the London Mathematical Society*, 1936.
8. J. M. Wing, “Computational Thinking”, *Communications of the ACM* 49(3), 33–35, 2006.

PART I · CHAPTER 1.3

Algorithms

Writing a program does not begin at the keyboard. First it becomes clear what the problem asks, then the solution is put on paper step by step, and only at the end are these steps translated into a programming language. These notes walk through every stop on this road with small but complete examples. The first sections cover defining a problem by its input, output and test cases; writing the solution in pseudocode and tracing it by hand with a variable table; and translating pseudocode into Java. Then come the algorithms for sorting two and three numbers, factorial, Fibonacci, exponentiation, palindromes and primes. Each is treated in the same pattern: problem definition, pseudocode, hand trace, Java code and tests. The notes end with the three simple methods of searching and sorting: random, linear and binary search; shuffling, selection sort and bubble sort. In every section, Java-specific pitfalls are given together with the real messages of the compiler and the runtime.

LEARNING OUTCOMES

Problem solving
Define a problem by its input, output and test cases; think about edge cases separately; write the solution first as an algorithm, then as code.

Pseudocode and tracing
Express an algorithm in pseudocode and trace it by hand, step by step, with a variable table.

Translating to Java
Translate reading, writing, decision and loop steps into their Java counterparts; use details such as `Scanner`, `print` / `println`, integer division, `long` and `equals` correctly.

Basic algorithms
Write, trace and test the algorithms for ordered printing, factorial, Fibonacci, palindromes, primes and exponentiation.

Searching
Compare random, linear and binary search; explain why binary search needs sorted data; compute the worst-case number of comparisons.

Sorting
Apply shuffling, selection sort and bubble sort step by step on an array; find the numbers of comparisons and swaps.

1.3.1 Problem solving

Someone new to programming often starts writing code the moment they read the problem. For small problems this works for a while. Once the problem grows a little, the code turns into a heap of patches whose purpose is unclear. When an error appears, its source cannot be found either, because nothing written down says "what the program is supposed to do". Experienced programmers therefore split the work into stages. The output of each stage is the input of the next, and each stage can be checked on its own.

1.3.1.1 Algorithm first, implementation second

Solving a problem with a computer is a cycle of four steps (Figure 1.3.1). The first step is **defining the problem**: what will be given and what is wanted? As long as the answer to this question stays vague, all the later steps can go to waste. The second step is **determining the algorithm**. An algorithm is the list of steps to follow to solve the problem. There may be more than one algorithm for the same problem; the right choice determines the speed and memory use of the program. The third step is **translating the algorithm into code** in the chosen programming language, in this course Java. In the fourth step the program is tried against **test cases** prepared beforehand. If a test does not give the expected result, we return to the earlier steps. The mistake may be in the coding, in the algorithm, or even in a misunderstanding of the problem.

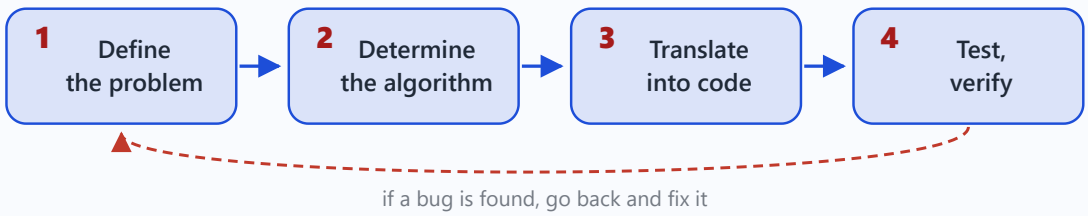


Figure 1.3.1: The problem-solving cycle. If a test shows an error, we go back along the dashed arrow and fix the step that was the source of the error.

The most important gain of this arrangement is that mistakes are caught while they are cheap. A mistake found in an algorithm on paper is fixed with a few lines. If the same mistake is found after it has been buried in a program of hundreds of lines, both finding and fixing it cost far more. Preparing the test cases before writing the code rests on the same reason. Tests prepared after the program is written tend, without our noticing, to treat what the program does as "correct".

1.3.1.2 What is an algorithm?

> **Definition 1.3.1**
>
> An **algorithm** is a sequence of instructions for solving a problem, made of clear steps in a definite order and finite in number.

The word comes from the name of the mathematician al-Khwarizmi, who lived in 9th-century Baghdad. The Latin spelling of his name was Algoritmi; the Latin translation of his book on calculating with decimal numbers became known in Europe under this name. Over time the word "algorithm" came to be used for any step-by-step described method of calculation.

A computer does not understand an algorithm directly. An algorithm is written in the way people think; a computer only runs commands in its own machine language. The bridge between them is the programming language: the algorithm is translated into a language such as Java, and the compiler turns this text into a form the machine can run. Just as an algorithm can be written as several programs, a problem can have several algorithms. These algorithms are usually compared by two measures: how long they take to finish and how much memory they use.

The most familiar everyday analogy is a cooking recipe. The ingredients are the input, the steps of preparation are the algorithm, and the resulting dish is the output. But recipes are written for people and contain vague steps such as "add a little salt" or "stir until it thickens". An experienced cook fills these gaps with their own knowledge. A computer cannot attach a number to the word "little"; it must be told "add 5 grams of salt". For a list of instructions to count as an algorithm, it is expected to have five properties (Table 1.3.1).

Property	Meaning	Where the recipe breaks it
Input	Takes zero or more initial data items.	—
Output	Produces at least one result.	—
Definiteness	Each step is clear and has a single meaning.	"Add a little salt."
Finiteness	Ends after a finite number of steps.	"Stir until it looks done."
Effectiveness	Each step can really be carried out.	"Chop infinitely fine."

Table 1.3.1: The five properties of an algorithm and their counterparts in a recipe.

Defining the input property as "zero or more" is deliberate. An algorithm that prints the first ten primes on the screen has no input at all, but it is still an algorithm. A method with no output, on the other hand, produces nothing and is useless; that is why at least one output is required. Finiteness means that the algorithm stops at some point for every input. A loop that runs forever when the user enters a wrong value breaks this property. Effectiveness requires the steps to be actually doable: the step "list all the primes" is clear and single-meaning, but cannot be done because there are infinitely many primes.

1.3.1.3 Defining the problem and test cases

Defining the problem means writing down the program's inputs and outputs explicitly in natural language. The sentence "add two numbers" looks sufficient at first glance. A little thought raises questions: are the numbers integers, or can they be decimals too? Can a negative number be entered? Will the result be written to the screen or used in another computation? As the answers to these questions are added to the definition, what the program will do becomes clear.

The second part of the definition is the test cases. A **test case** is writing down in advance the expected output for a given input. The program is run and the output it produces is compared with the expected one; if the two are the same the test passes (Figure 1.3.2). The expected output must be found independently of the program, by hand or in another reliable way. Otherwise the test amounts to nothing more than comparing the program with itself.

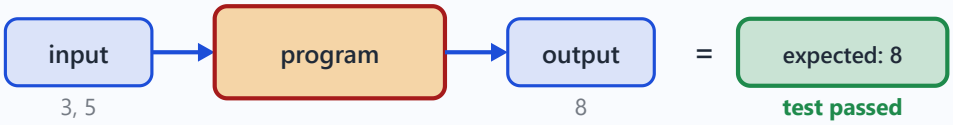


Figure 1.3.2: How a test works: the input 3 and 5 is given to the program, the 8 it produces is compared with the expected 8, and the test passes.

Table 1.3.2 shows test cases prepared for two small problems. In each problem the first two rows are ordinary inputs. The third rows are deliberately chosen **edge cases**. Zero, an empty string, a negative number, the first or last element of an array are extreme inputs of this kind. Programmers naturally think of ordinary inputs; extremes are easily forgotten. That is why bugs hide most often at the extremes. A program that counts words may give 2 for "Hello, world!" yet give 1 for an empty string. A simple approach that splits the text at spaces counts even an empty string, which contains no spaces, as one piece.

Problem	Test	Input	Expected output
Sum of two numbers	1	3, 5	8
	2	−2, 7	5
	3 (edge)	0, 0	0
Number of words in a text	1	"Hello, world!"	2
	2	"A short test."	3
	3 (edge)	"" (empty string)	0

Table 1.3.2: Test cases for two problems. The third rows are edge cases.

A good test set covers few but mutually different cases. For addition, the tests 3 + 5, 4 + 6 and 10 + 20 test the same thing three times: all three are positive integers. Choosing instead a positive pair, a pair with a negative and a pair with a zero gives much more chance of catching errors for the same effort.

In all the algorithms in these notes the test cases are chosen with this principle: each has both ordinary and extreme inputs.

1.3.1.4 Determining the algorithm and tracing by hand

Once the problem is defined, the steps of the solution are written in order. For the sum of two numbers, the algorithm has four steps:

1. Get the first number.
2. Get the second number.
3. Add the two numbers.
4. Print the sum on the screen.

The job is not done once the algorithm is written; before any code is written, the algorithm itself is tested. The way to do this is the **dry run**. A test case is chosen and the algorithm is run on paper step by step. After each step the values of the variables are written into a table. This table is called a **trace table**. The columns of the table are the variables and the screen, and the rows are the steps. If a cell is left empty, that variable has not yet received a value at that moment.

Step	num1	num2	sum	Screen
1. Get the first number	3			
2. Get the second number	3	5		
3. Add the two numbers	3	5	8	
4. Print the sum	3	5	8	8

Table 1.3.3: Dry run of the addition algorithm for test 1 (input 3, 5). 8 was printed on the screen; the expected value was also 8.

In this example the trace is very easy and may seem unnecessary. But the same habit makes a big difference with loop algorithms. How many times a loop runs, what value a variable takes in which iteration, whether the loop ends one iteration too early or too late: these can be seen reliably only with a table like this. For hand tracing, small but meaningful test cases are chosen. For factorial, 5 is a good choice: five iterations can be traced by hand and the result (120) is known by heart. 20 is a bad choice; both the table grows long and verifying the result becomes difficult.

1.3.1.5 Pseudocode

An algorithm can be written in four forms: in natural language, in pseudocode, as a flowchart, or directly in a programming language. Natural language is a tool everyone understands but is open to ambiguity. A programming language is exact but its burden of detail is heavy: semicolons, type declarations and library names overshadow the real idea. **Pseudocode** stands between the two.

Definition 1.3.2

Pseudocode is a notation for algorithms, written for people, that resembles a programming language but has no compiler rules. It uses fixed keywords for basic structures such as reading, writing, assignment, decisions and loops; everything else can be written in natural language.

Pseudocode has two big benefits. First, it makes it easier to find logic errors before writing code: attention goes not to syntax but to the order of the steps and the conditions. Second, it lets programmers who use different languages talk about the same algorithm. Someone who knows Java and someone who knows Python can read the same pseudocode and translate it into their own languages. The pseudocode keywords used in these notes and their Java counterparts are collected in Table 1.3.4.

Pseudocode	Meaning	Java counterpart
Start … End	Beginning and end of the algorithm	The `{ … }` body of the `main` method
Read x	Get a value from the user	`x = scanner.nextInt();`
Print "…", x	Display on the screen	`System.out.println("…" + x);`
x = 5	Assign a value to a variable	`x = 5;`
If x = y then	Compare for equality	`if (x == y)` (two equals signs)
Else	If the condition is false	`else`
For i = 1 to n	Repeat	`for (int i = 1; i <= n; i++)`
Break	Leave the loop at once	`break;`
%, and, or	Remainder, logical and/or	`%`, `&&`, `||`

Table 1.3.4: Pseudocode keywords and their Java counterparts.

The trickiest row of the table is the equality row. In pseudocode and in mathematics the `=` sign is used both for assignment and for comparison; the meaning is inferred from context. In Java these two jobs are done with different signs. A single equals sign (`=`) puts the value on the right into the variable on the left. A double equals sign (`==`) compares two values and produces `true` or `false`. When translating pseudocode into Java, the sign in every `If … = … then` line must be turned into `==`.

Common mistake

Using a single equals sign in a condition. The line `if (num % divisor = 0)` does not compile. Since the left side is a computed value and not a variable, the compiler gives this message: `error: unexpected type … required: variable, found: value`. If the left side were a variable, for example `if (s = 5)`, the message would change: `error: incompatible types:`

`int cannot be converted to boolean`. Since `if` accepts only `boolean` in Java, this mistake is caught at compile time. In languages like C the same line compiles silently and corrupts the logic of the program.

A similar distinction exists for logical operators. The "and" of pseudocode is written `&&` in Java and "or" is written `||`. These operators short-circuit. In `a && b`, if `a` is false the result is false anyway, so `b` is never evaluated. In `a || b`, if `a` is true, `b` is not looked at.

1.3.2 Translating to code

After the algorithm is prepared and traced by hand, the next step is coding (*implementation*). The programming language expresses the algorithm in a form the computer can compile and run. When writing code, the program is divided into logical parts: taking input, computing and giving output. In larger programs each part moves into a separate method. Each line of pseudocode usually corresponds to one or a few lines of Java.

1.3.2.1 From pseudocode to a running program

There are four stops before an algorithm produces a result on screen (Figure 1.3.3). The pseudocode is translated into Java source code and written to a file such as `Addition.java`. The Java compiler `javac` reads this file. If there is no typing error in the code, it produces a bytecode file named `Addition.class`. Bytecode is not the machine language of a particular processor. It is an intermediate language understood by the Java Virtual Machine (JVM). The command `java Addition` starts the JVM; the JVM runs the bytecode and the output appears on screen. The same `.class` file runs on Windows, on Linux or on macOS, on any machine with a JVM installed.

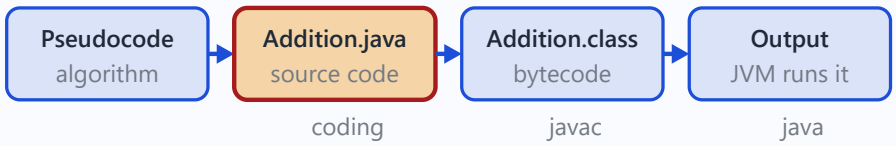


Figure 1.3.3: From pseudocode to output: coding, compiling with `javac`, and running on the JVM with the `java` command.

Errors can show up at two different moments. **Compile errors** are found while `javac` runs: a missing semicolon, a misspelled keyword, an undeclared variable, an incompatible type. No `.class` file is produced from a program with a compile error. **Runtime errors**, on the other hand, appear while the program runs: accessing a nonexistent cell of an array, dividing by zero, the user typing a letter instead of a number. The third and most dangerous kind is **logic errors**. The program compiles, runs, gives no message, but produces a wrong result. Only tests catch such errors.

The correspondence between the algorithm steps and the Java lines is seen in Table 1.3.5. Each step has become a single line. In more complex algorithms a step may take several lines; but the correspondence must always remain traceable.

Algorithm step	Java
Get the first number	`num1 = scanner.nextDouble();`
Get the second number	`num2 = scanner.nextDouble();`
Add the two numbers	`sum = num1 + num2;`
Print the sum	`System.out.println(sum);`

Table 1.3.5: The steps of the addition algorithm and their Java counterparts.

A note on naming: the Java libraries use English names (`Scanner`, `nextDouble`, `println`), and in this course we choose our own variable and method names in English too, and keep them to plain ASCII letters. Java does accept names with letters from other alphabets, such as the Turkish ş or ı. Still, when a source file is opened with a different character encoding, such letters can get garbled and lead to a compile error. Pseudocode has no such problem; there, names like `firstInteger` can be used freely.

1.3.2.2 The addition program

The complete Java program for the addition algorithm is below. Code that the slides show only as the body of `main` is given here together with the `import` line and the class definition; in this form it can be written to a file named `Addition.java` and compiled.

```
import java.util.Scanner;

public class Addition {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the first number: ");
        double num1 = scanner.nextDouble();
        System.out.print("Enter the second number: ");
        double num2 = scanner.nextDouble();
        double sum = num1 + num2;
        System.out.println(num1 + " + " + num2
                + " = " + sum);
        scanner.close();
    }
}
```

The `import java.util.Scanner;` on the first line tells the compiler where to find the `Scanner` class. Java's standard library is divided into packages, and `Scanner` is in the `java.util` package. A few basic classes such as `System` and `String` are in the `java.lang` package, which is added to every

program automatically. Every other class needs an `import`. If the line is forgotten, the compiler does not recognize the name `Scanner`:

```
Addition.java:4: error: cannot find symbol
        Scanner scanner = new Scanner(System.in);
        ^
  symbol:   class Scanner
  location: class Addition
```

Since `Scanner` appears twice on the same line, the compiler reports this error twice and says "2 errors". A single missing line can cause several error messages. This is why compile errors are always read starting from the first one.

The line `public class Addition` defines the class. The name of a public class must be the same as the file name: the `Addition` class sits in the file `Addition.java`. `public static void main(String[] args)` is the starting point of the program; the JVM begins running from this method. The counterpart of the `Start … End` pair in the pseudocode is the curly braces of this method.

`Scanner scanner = new Scanner(System.in);` creates an object that will read from the keyboard. `System.in` is the standard input stream, that is, the keyboard. The `nextDouble()` method of the `scanner` object reads the next number as a `double`, `nextInt()` as an `int`, and `nextLine()` reads the whole line as text. The `scanner.close();` at the end of the program releases the resource the object uses.

In the line `double sum = num1 + num2;`, the right side is computed first and then the result is put into the variable on the left. Assignment always works from right to left. That is why a line like `total = total + 1`, which looks absurd in mathematics, makes sense in programming: "add 1 to the current value of total and put the result back into total".

1.3.2.3 Tracing the addition program

The line-by-line trace of a run in which the user enters 3 and 5 is in Table 1.3.6. The line numbers are the lines of the `main` body: line 1 is where the `Scanner` is created, and line 9 is the `close` call.

Line	num1	num2	sum	Screen	Explanation
1	—	—	—		A `Scanner` is created to read from the keyboard.
2	—	—	—	Enter the first number:	`print` does not move to the next line; the cursor waits next to the message.
3	3.0	—	—	… number: 3↵	The user types 3 and presses Enter; `nextDouble()` reads 3.0.
4	3.0	—	—	Enter the second number:	The second message is printed.
5	3.0	5.0	—	… number: 5↵	The user enters 5.
6	3.0	5.0	8.0		The right side is computed: 3.0 + 5.0 = 8.0.

Line	num1	num2	sum	Screen	Explanation
7–8	3.0	5.0	8.0	3.0 + 5.0 = 8.0↵	`+` concatenates the strings.
9	3.0	5.0	8.0		The scanner is closed and the program ends.

Table 1.3.6: Trace of the addition program (input 3 and 5). The "Screen" column is the text accumulated on screen after that line; ↵ is the end of a line.

The last line the program writes to the screen is "3.0 + 5.0 = 8.0". The test case expected 8; the program printed 8.0. The value is the same; the form of writing differs. When a number of type `double` is converted to text, it is written with its decimal part, as "8.0", even if it is a whole number. Such format differences must also be considered when writing test cases. A program that does the testing automatically counts the texts "8" and "8.0" as different.

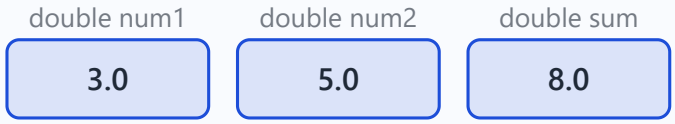


Figure 1.3.4: The three `double` variables in memory just before the program ends.

1.3.2.4 Details of reading and writing

`print` and `println`. Both methods write the value given to them on the screen. The difference is in what happens afterwards. `println` (*print line*) writes the value and moves to the next line; `print` does not. This is why `print` is used when asking the user for input: the cursor waits right next to the message "Enter the first number: " and the user types the number on the same line. `print` is also used to write a series of values side by side on one line; after the last value a `println()` completes the line.

String concatenation and order of evaluation. The `+` operator adds between two numbers and concatenates between a string and any value. An expression is evaluated from left to right, and this can lead to surprising results. Table 1.3.7 shows outputs obtained by running on JDK 21.

Expression	Output	Reason
`3 + 5 + " total"`	8 total	First 3 + 5 = 8 is added, then it is appended to the string.
`"total " + 3 + 5`	total 35	String + 3 = "total 3"; then + 5 is concatenation again.
`"total " + (3 + 5)`	total 8	The parentheses bring the addition forward.
`7 / 2`	3	Integer division between two `int`s; the fraction is discarded.
`7 % 2`	1	Remainder.
`-7 / 2` and `-7 % 2`	−3 and −1	The quotient is truncated toward zero; the remainder takes the sign of the dividend.

Expression	Output	Reason
`7.0 / 2`	3.5	If one of the operands is a `double`, true division is performed.

Table 1.3.7: Behavior of the `+` operator with numbers and strings; division and remainder.

Integer division is used often in the algorithms of this chapter. The `num / 2` in the prime check and the `(low + high) / 2` in binary search are both integer division and discard the fractional part on purpose. The remainder operator `%` is at least as important. `num % divisor == 0` means "divides exactly"; `num % 10` gives the last digit, and `num % 2` tells whether a number is even or odd.

Decimal separator and locale. When reading numbers, `Scanner` uses the computer's locale. On a computer set to Turkish, the decimal separator is the comma. So typing "3,5" into the addition program works, while typing "3.5" stops the program:

```
Enter the first number: 3.5
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:947)
        at java.base/java.util.Scanner.next(Scanner.java:1602)
        at java.base/java.util.Scanner.nextDouble(Scanner.java:2573)
        at Addition.main(Addition.java:7)
```

On a computer set to English the situation is the reverse: "3.5" is accepted and "3,5" is rejected. The output of the program, however, is in the form "3.5" in both settings, because converting a `double` value to text does not look at the locale. The same program behaving differently on different computers often surprises students. If you want reading to be the same everywhere, create the scanner as `new Scanner(System.in).useLocale(Locale.US)`. When the user types a letter instead of a number, the same `InputMismatchException` is thrown. Catching this exception and asking the user again is a more advanced topic. The programs in these notes assume that the user enters a valid number.

1.3.3 Algorithms with decision structures

From this section on, every algorithm is treated in the same pattern. First the problem and test cases are defined, then pseudocode is written and traced by hand with a variable table. Finally the Java code is given and the tests are run. The first two examples contain no loops. The path the algorithm follows depends on decisions made by comparing the inputs.

1.3.3.1 Printing two integers in order

Problem definition. The input is two integers. The output is the same two numbers, with the smaller one written first. The test cases are in Table 1.3.8. The third test is an edge case: the two numbers are equal.

Input	Expected	Case tested
4, 11	4 11	Numbers already in order
13, 11	11 13	Numbers in reverse order
7, 7	7 7	Equal numbers (edge)

Table 1.3.8: Test cases for printing two integers in order.

Pseudocode. The algorithm reads two numbers and makes a single comparison. If the first number is smaller than the second, the numbers are printed in the order they were read; otherwise in the reverse order.

```
Start
    Print "Enter the first integer: "
    Read firstNum
    Print "Enter the second integer: "
    Read secondNum
    If firstNum < secondNum then
        Print firstNum, " ", secondNum
    Else
        Print secondNum, " ", firstNum
    End If
End
```

The `If … then … Else … End If` structure splits the algorithm into two branches. If the condition is true only the first branch runs, if false only the second; the two branches never run together. After the `End If` line the two paths merge again.

Hand trace. Table 1.3.9 is the trace of the first two tests. For the input (4, 11) the condition is true and the `then` branch runs. For the input (13, 11) the condition is false; the `Else` branch first writes the second number, that is, the smaller one.

Test	Step	firstNum	secondNum	Condition	Screen
1	Read firstNum	4			
	Read secondNum	4	11		
	If 4 < 11	4	11	true	
	`then` branch	4	11		4 11
2	Read firstNum	13			
	Read secondNum	13	11		
	If 13 < 11	13	11	false	
	`Else` branch	13	11		11 13

Table 1.3.9: Hand trace of the two-integer algorithm for (4, 11) and (13, 11).

The equal-numbers test can be settled even without tracing. For (7, 7) the condition 7 < 7 is false and the `Else` branch runs. This branch writes "second, then first"; since the two numbers are the same, the output is again "7 7". Had `≤` been used in the condition instead of `<`, the `then` branch would have run for equal numbers and the result would not change. The edge-case test catches no bug here but guarantees that the algorithm is correct for equal numbers too.

Java code. The Java counterpart of the pseudocode is below. Each `Print` became a `System.out.print` or `println`, and each `Read` a `nextInt()`.

```
import java.util.Scanner;

public class SortTwo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the first integer: ");
        int first = scanner.nextInt();
        System.out.print("Enter the second integer: ");
        int second = scanner.nextInt();
        if (first < second) {
            System.out.println(first + " " + second);
        } else {
            System.out.println(second + " " + first);
        }
        scanner.close();
    }
}
```

When the program is compiled with JDK 21 and run with the three tests, it prints "4 11", "11 13" and "7 7" in turn; all three tests pass. At the start of the output the messages "Enter the first integer: Enter the second integer: " also appear. When the user types the numbers in a terminal, these messages fall on separate lines because the user presses Enter after each number.

In Java each branch of `if` and `else` is either a single statement or a block wrapped in curly braces. In the code above, even the one-line branches are put in braces. This is not a requirement but a good habit. If a second line is later added to a brace-less branch, that line always runs regardless of the condition, and its indentation hides this.

1.3.3.2 Printing three integers in order

Problem definition. The input is three integers; the output is the same numbers written from smallest to largest. This problem makes you think about choosing test cases. Three numbers can arrive in 3! = 6 different orders: 1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 1 2 and 3 2 1. Each order passes through a different path of the algorithm; if all six orders are tried, every branch of the algorithm will have run at least once. Inputs with equal numbers are added to these: for 5 5 2 the expected output is 2 5 5.

Idea. First the smallest of the three numbers is found and written. The remaining two numbers are sorted and written with the two-integer algorithm. Since the smallest number can be the first, second or third, the algorithm has three main branches. Within each main branch there is also a small two-number decision.

```
Start
    // Get the first integer
    Print "Enter the first integer: "
    Read firstNum
    // Get the second integer
    Print "Enter the second integer: "
    Read secondNum
    // Get the third integer
    Print "Enter the third integer: "
    Read thirdNum
    // Print in ascending order
    If firstNum ≤ secondNum and
         firstNum ≤ thirdNum then
        Print firstNum
        If secondNum ≤ thirdNum then
            Print secondNum, " ", thirdNum
        Else
            Print thirdNum, " ", secondNum
        End If
    Else If secondNum ≤ firstNum and
         secondNum ≤ thirdNum then
        Print secondNum
        If firstNum ≤ thirdNum then
            Print firstNum, " ", thirdNum
        Else
            Print thirdNum, " ", firstNum
        End If
    Else
        Print thirdNum
        If firstNum ≤ secondNum then
            Print firstNum, " ", secondNum
        Else
            Print secondNum, " ", firstNum
        End If
    End If
End
```

The first condition asks whether the first number is less than or equal to both of the others. The two comparisons are joined with "and": both must be true. If this branch runs, the first number is printed, and the remaining second and third numbers are ordered with a nested decision. If the first condition is false, we check whether the second number is the smallest. If that is false too, the smallest number is the third and the last branch runs. No separate condition is needed in the last branch: once the first two are ruled out, only one possibility is left.

Using `≤` rather than `<` in the comparisons is necessary for equal numbers. Suppose the conditions were written with strict `<` and the input were 5 5 7. For the first number, 5 < 5 is false; for the second number, 5 < 5 is false too. The algorithm falls into the last branch, writes 7 first as the "smallest" number, and the output is "7 5 5". When the smallest value occurs in two numbers at once, no number is "strictly smaller than the others". `≤`, on the other hand, directly expresses "the smallest or one of the smallest", and for this input it runs the first branch. The test 5 5 2 does not catch this bug, because there the smallest number is unique. Only a test like 5 5 7, where the smallest value is repeated, exposes it.

An observation is hidden in the pseudocode. The same job of "sort two numbers and print them" is repeated in three places, with only the variable names changed. A repeated job is a very good candidate for turning into a function (a method in Java). If a method like `printTwo(int a, int b)` is written, each of the three branches shrinks to a single line. Methods are a separate topic; what matters here is the habit of noticing repetition.

1.3.3.3 Java code and trace for three integers

The full Java program from the slides is below. The three main branches of the pseudocode have become an `if … else if … else` chain, and the inner decisions one-line `if … else` statements. The "and" has become `&&`.

```
import java.util.Scanner;

public class SortThree {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter three integers: ");
        int first = scanner.nextInt();
        int second = scanner.nextInt();
        int third = scanner.nextInt();
        if (first <= second && first <= third) {
            System.out.print(first + " ");
            if (second <= third) System.out.println(second + " " + third);
            else                 System.out.println(third + " " + second);
        } else if (second <= first && second <= third) {
            System.out.print(second + " ");
            if (first <= third) System.out.println(first + " " + third);
            else                System.out.println(third + " " + first);
        } else {
            System.out.print(third + " ");
            if (first <= second) System.out.println(first + " " + second);
            else                System.out.println(second + " " + first);
        }
        scanner.close();
    }
}
```

There is one more detail in this program. `System.out.print("Enter three integers: ")` writes a single message and the three `nextInt()` calls follow one after another. `Scanner` reads numbers as pieces separated by spaces or line ends. The user can type "7 3 5" and press Enter once, or press Enter after each number; in both cases the three numbers are read correctly.

Code trace. Let the input be 7, 3, 5. Table 1.3.10 gives the value of each condition and what is printed, step by step.

Step	Expression examined	Value	Screen
1	`first <= second` → 7 ≤ 3	false; the right side of `&&` is not looked at	
2	`second <= first && second <= third` → 3 ≤ 7 and 3 ≤ 5	true; the smallest is the second	
3	`System.out.print(second + " ")`	—	3
4	`second <= third` → 7 ≤ 5	false	3
5	`else` branch: `println(third + " " + first)`	—	3 5 7

Table 1.3.10: Trace of the `SortThree` program for (7, 3, 5). first = 7, second = 3, third = 5.

Short-circuit behavior shows in the first step. As soon as `first <= second` turns out false, it is certain that the whole `&&` expression will be false; Java never makes the comparison `first <= third`. In this example the skipped comparison is harmless. But when the right side is a method call or an expression that risks division by zero, short-circuiting is used on purpose. The expression `b != 0 && a / b > 2` never attempts the division when `b` is zero.

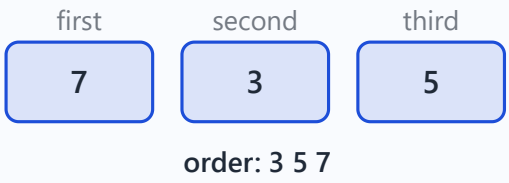


Figure 1.3.5: The three variables at the end of the trace and the order produced: 3 5 7.

Tests. The program prints "1 2 3" for each of the six orders. For 5 5 2 it prints "2 5 5" and for 7 3 5 it prints "3 5 7". These results were obtained by running the program itself on JDK 21. The inputs 5 2 5, 2 5 5, 5 5 7 and 4 4 4 were also tried, and the right order was found in all. When the same program was rewritten with strict `<` and tried, it printed "7 5 5" for 5 5 7.

Common mistake

Trying to solve the three-number problem by writing out all the possibilities one by one. A separate condition can be written for each of the six orders: `if (a < b && b < c) … else if (a < c && c < b) …`. The result is a six-branch structure that is hard to read. For equal

numbers, some of the conditions may enter no branch and the program ends without printing anything. The idea "first find the smallest, then sort the rest" breaks the problem into small parts and makes it possible to verify each part separately.

1.3.4 Algorithms with loops

Decision structures determine which path the algorithm takes. Loops make a job repeat. The factorial, Fibonacci and exponentiation algorithms in this section rest on the same basic pattern. One or two variables get an initial value before the loop. In each iteration of the loop these variables are updated. When the loop ends, the result is in these variables. The details of the pattern vary from algorithm to algorithm, and mistakes occur precisely in those details.

1.3.4.1 Factorial

Definition 1.3.3

The **factorial** of a non-negative integer n is the product of all the integers from 1 to n: $n! = 1 \times 2 \times 3 \times \ldots \times n$. Also, $0! = 1$ by convention.

The rule $0! = 1$ may look arbitrary. But when no numbers are multiplied, what remains is 1, the identity of multiplication, just as the result is 0 when no numbers are added. The factorial gives the number of different orders in which n distinct objects can be arranged. That is why 3! = 6 orders were tried in the three-number sorting problem. Table 1.3.11 gives the test cases.

n	$n!$
0	1
1	1
3	6
5	120
10	3.628.800

Table 1.3.11: Test cases for factorial. 0 and 1 are edge cases.

Pseudocode.

```
Start
    // Enter a number
    Read num
    // Initialize the result
    factorial = 1
    // Loop to compute the factorial
    For i = 1 to num
        factorial = factorial * i
    End For
    Print "Factorial of the number: ", factorial
End
```

The `factorial` variable is an **accumulator**. In each iteration of the loop, a piece of the result is added onto it; here "adding" means multiplying. The accumulator's initial value is the identity of the operation being done: 1 for multiplication, 0 for addition. If the accumulator were initialized to 0, every product would be 0 and the result would always come out 0.

Hand trace. The trace for *num = 5* is in Table 1.3.12. The loop variable `i` starts at 1 and increases by one each iteration. When `i` becomes 6, the condition *6 ≤ 5* is false and the loop ends.

Iteration	`i`	`i ≤ 5`	Operation	`factorial`
start	—	—	`factorial = 1`	1
1	1	true	1 × 1	1
2	2	true	1 × 2	2
3	3	true	2 × 3	6
4	4	true	6 × 4	24
5	5	true	24 × 5	120
exit	6	false	the loop ends; 120 is printed	120

Table 1.3.12: Trace of the factorial algorithm for *num = 5*.

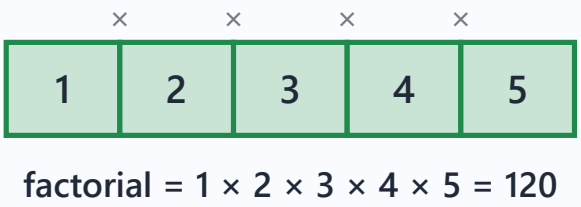


Figure 1.3.6: When the loop ends, all five factors have been used: 1 × 2 × 3 × 4 × 5 = 120.

In the edge case *num = 0*, the loop condition is false on the first try (*1 ≤ 0*); the body never runs and the accumulator's initial value 1 is printed. This agrees with the rule *0! = 1*. The algorithm needs no separate branch for 0; the right initial value solves the edge case by itself.

Java code.

```
import java.util.Scanner;

public class Factorial {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int num = scanner.nextInt();
        long factorial = 1;
        for (int i = 1; i <= num; i++) {
            factorial = factorial * i;
        }
        System.out.println("Factorial of the number: " + factorial);
        scanner.close();
    }
}
```

The line `For i = 1 to num` in the pseudocode has become `for (int i = 1; i <= num; i++)` in Java. The `for` header has three parts: the initialization (`int i = 1`), the continuation condition (`i <= num`) and the update done at the end of each iteration (`i++`). The "to num" in the pseudocode includes the number itself; that is why the condition is written with `<=` , not `<` .

Why `long` ? The factorial grows very fast. In Java, `int` is a 32-bit integer and holds at most 2,147,483,647. 12! = 479,001,600 is below this limit; 13! = 6,227,020,800 is above it. Had the accumulator been declared `int` , the program would print 1,932,053,504 for 13 without any error message. This number is obtained by subtracting $2^{32} = 4{,}294{,}967{,}296$ from 6,227,020,800: when the `int` limit is exceeded, the number wraps around. This event is called **overflow**. Java does not check for overflow and throws no exception. `long` is 64 bits and holds at most 9,223,372,036,854,775,807; 20! = 2,432,902,008,176,640,000 fits in it. For 21!, `long` overflows too and the program prints a meaningless negative number such as −4,249,290,049,419,214,848. For larger factorials the `java.math.BigInteger` class is used.

Common mistake

Initializing the accumulator inside the loop. If the line `long factorial = 1;` is written inside the `for` body, the variable becomes 1 again in every iteration. After the loop the variable cannot even be reached, because a variable declared in the body is valid only in that body. The compiler reports a `cannot find symbol` error at the `System.out.println` line. The accumulator is always declared before the loop, outside it.

1.3.4.2 The Fibonacci sequence

Definition 1.3.4

The first two terms of the **Fibonacci sequence** are 0 and 1; every later term is the sum of the two terms before it: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

The problem is to print as many terms as the number n that the user enters. The test cases are in Table 1.3.13. $n = 1$ and $n = 2$ are edge cases: in these cases the loop should not run at all, and only the first terms should be printed.

n (number of terms)	Expected output
1	0
2	0 1
5	0 1 1 2 3
7	0 1 1 2 3 5 8

Table 1.3.13: Test cases for Fibonacci.

```
Start
    // Enter how many terms are wanted
    Read n
    // Initialize the first two Fibonacci numbers
    a = 0
    b = 1
    If n ≥ 1 then Print a
    If n ≥ 2 then Print b
    // Compute the remaining terms
    For i = 2 to n - 1
        result = a + b
        a = b
        b = result
        Print result
    End For
End
```

The algorithm does not store the whole sequence; it keeps only the last two terms. `a` is the one before last, `b` is the last term. The new term is computed with `result = a + b`. Then two assignments shift the values by one step: `a = b` makes the old last term the "previous", and `b = result` makes the new term the "last". The first two terms are written before the loop, after checking whether n is large enough. The loop runs from $i = 2$ to $n - 1$, that is, $n - 2$ iterations. Since each iteration prints one term, the total number of terms is $2 + (n - 2) = n$.

Hand trace. The trace for $n = 5$ is given line by line in Table 1.3.14.

Stage	`i`	`result`	`a`	`b`	Screen
start	—	—	0	1	
5 ≥ 1, 5 ≥ 2	—	—	0	1	0 1
iteration 1	2	0 + 1 = 1	1	1	0 1 1
iteration 2	3	1 + 1 = 2	1	2	0 1 1 2
iteration 3	4	1 + 2 = 3	2	3	0 1 1 2 3
exit	5	5 ≤ 4 false; the loop ends			0 1 1 2 3

Table 1.3.14: Trace of the Fibonacci algorithm for *n = 5*.

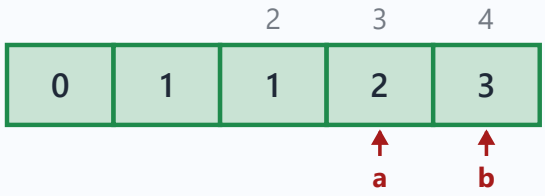


Figure 1.3.7: The five terms computed when the loop ends; `a` and `b` show the last two terms.

The order of the shift. The order of the lines `a = b;` and `b = result;` matters. If the order is swapped and `b = result` is written first, the old value of `b` is lost and the line `a = b` puts the new term into `a`. In that case `a` and `b` become equal and the sequence turns into one that doubles, like 0 1 1 2 4 8 16 .... In assignments, the variable whose value is going to be overwritten is always updated last.

Java code.

```java
import java.util.Scanner;

public class Fibonacci {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("How many terms? ");
        int n = scanner.nextInt();
        long a = 0, b = 1, result;
        if (n >= 1) System.out.print(a + " ");
        if (n >= 2) System.out.print(b + " ");
        for (int i = 2; i <= n - 1; i++) {
            result = a + b;
            a = b;
            b = result;
            System.out.print(result + " ");
        }
        System.out.println();
        scanner.close();
    }
}
```

The Java code overlaps the pseudocode line by line. The only difference is that the term variables are chosen as `long`. Fibonacci terms grow exponentially too. With `int`, F(46) = 1,836,311,903 is the last correct term. The true value of F(47) is 2,971,215,073 and exceeds the `int` limit; the program prints −1,323,752,223 instead. Counting the sequence from 0, with F(0) = 0 and F(1) = 1, F(47) is the 48th term of the sequence. So the `int` program works correctly up to *n = 47*; when *n = 48*, the last term it writes is corrupted. With `long`, correct values are obtained up to F(92). The program prints "0 1 1 2 3 5 8 13" for *n = 8* and "0 1 1 2 3 5 8 13 21 34" for *n = 10*.

Common mistake

An off-by-one error in the loop bound. If the Fibonacci loop is written with `i <= n` instead of `i <= n - 1`, the loop runs one iteration too many and *n + 1* terms are printed: for *n = 5*, 0 1 1 2 3 5. Mistakes like this are found most easily by tracing a small test by hand. *n = 1* and *n = 2* should also be checked separately: in these two cases the loop must not run at all.

1.3.4.3 Exponentiation

The power of a number is multiplying the same number by itself repeatedly: $base^{exp} = base \times base \times \ldots \times base$ (exp times). Also, $base^{0} = 1$. The problem is to compute this value for a non-negative integer exponent. Table 1.3.15 gives the test cases; the row where the exponent is 0 is an edge case.

base	exp	Expected
3	4	81

base	exp	Expected
2	10	1024
5	0	1

Table 1.3.15: Test cases for exponentiation.

```
Start
    // Enter a number
    Read base
    // Enter an exponent
    Read exp
    // Initialize the result
    result = 1
    // Compute the result using the exponent
    For i = 1 to exp
        result = result * base
    End For
    Print "Result: ", result
End
```

This algorithm follows the same pattern as the factorial: the accumulator starts at 1 and the loop does one multiplication each iteration. The difference is that the multiplier is the same in every iteration. In the factorial the multiplier is the loop variable `i`; here it is always `base`. `i` is used only to count iterations. When the exponent is 0, the loop condition ($1 \le 0$) is false from the start, the body never runs and the result stays 1.

Iteration	`i`	Operation	`result`
start	—	`result = 1`	1
1	1	1 × 3	3
2	2	3 × 3	9
3	3	9 × 3	27
4	4	27 × 3	81
exit	5	5 ≤ 4 false; "Result: 81" is printed	81

Table 1.3.16: Trace of the exponentiation algorithm for base = 3, exp = 4.

Table 1.3.16 shows the four iterations for base = 3, exp = 4. The loop runs exactly exp times; that is, the number of multiplications the algorithm does equals the exponent. For 2^{10}, 10 multiplications are made and the result is 1024.

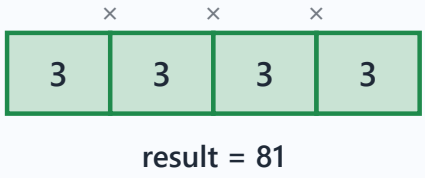


Figure 1.3.8: 3^4: four multiplications by 3, result 81.

Java code.

```
import java.util.Scanner;

public class Power {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Base: ");
        int base = scanner.nextInt();
        System.out.print("Exponent: ");
        int exp = scanner.nextInt();
        long result = 1;
        for (int i = 1; i <= exp; i++) {
            result = result * base;
        }
        System.out.println("Result: " + result);
        scanner.close();
    }
}
```

Java has no exponent operator. Writing `2 ^ 10` compiles but does not exponentiate: `^` is the bitwise "exclusive or" operator, and the value of the expression `2 ^ 10` is 8. A ready-made way is the `Math.pow(base, exp)` method; however, this method returns a `double`. `Math.pow(3, 4)` gives 81.0, not 81, and with very large integers the last digits may be wrong because of the limited precision of `double`. For integer powers the loop above is a reliable way. Overflow applies here too: had `result` been `int`, 2^{31} would give −2,147,483,648 and 2^{32} would give 0. With `long` the program prints the correct value for 2^{31}, 2,147,483,648.

1.3.5 Text processing and the flag

The two algorithms in this section bring new tools. The palindrome check works with text rather than numbers and raises the question of how text is compared in Java. The prime check introduces the flag pattern, which gathers the result of a loop in a single boolean variable, and leaving a loop early.

1.3.5.1 Palindrome check

Definition 1.3.5

A **palindrome** is a word or sentence that reads the same backwards as forwards: kayak, abcba, "A man, a plan, a canal: Panama". In sentences, differences of case, spaces and punctuation marks are ignored.

The last sentence of the definition adds a preprocessing step to the algorithm. When the sentence "A man, a plan, a canal: Panama" is reversed letter by letter, it becomes "amanaP :lanac a ,nalp a ,nam A", which is not the same as the original. Capital letters, spaces and punctuation break the comparison. So the text is first **cleaned**: all letters are converted to lowercase, and characters other than letters and digits are discarded. The cleaned text is "amanaplanacanalpanama", and this text really is equal to its reverse.

Input	Clean text	Expected
abcba	abcba	palindrome
Kayak	kayak	palindrome
Hello	hello	not
A man, a plan, a canal: Panama	amanaplanacanalpanama	palindrome

Table 1.3.17: Test cases for the palindrome check.

The "Kayak" test checks that capital letters are cleaned, the "Hello" test that an input that is not a palindrome is correctly rejected, and the last test that spaces and punctuation are discarded (Table 1.3.17). The pseudocode has three steps: clean, reverse, compare.

```
Start
    // Ask for a word or sentence
    Read inputText
    // Clean: lowercase, no spaces or punctuation
    cleanText = Clean(inputText)
    // Reverse the cleaned input
    reversed = Reverse(cleanText)
    // Compare cleaned and reversed input
    If cleanText = reversed then
        Print "The text is a palindrome."
    Else
        Print "The text is not a palindrome."
    End If
End
```

`Clean` and `Reverse` are written in the pseudocode as one-line operations. Each of them is really a small algorithm in its own right. This flexibility of pseudocode is useful: the main idea is set up first and the details of the sub-jobs are left for later.

Hand trace. The trace for the inputs "abcba" and "Hello" is in Table 1.3.18.

Step	“abcba”	“Hello”
Read inputText	abcba	Hello
cleanText = Clean(…)	abcba (no change)	hello
reversed = Reverse(…)	abcba	olleh
cleanText = reversed ?	true	false (first letter h ≠ o)
Screen	The text is a palindrome.	The text is not a palindrome.

Table 1.3.18: Trace of the palindrome algorithm for two tests.

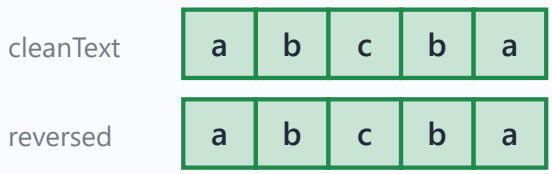


Figure 1.3.9: For "abcba", the clean text and its reverse match letter by letter.

Java code. In the following program the cleaning job is moved into a separate method named `clean`. For reversing, Java's ready-made `StringBuilder` class is used.

```
import java.util.Locale;
import java.util.Scanner;

public class Palindrome {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a word or sentence: ");
        String inputText = scanner.nextLine();
        String cleanText = clean(inputText);
        String reversed = new StringBuilder(cleanText).reverse().toString();
        if (cleanText.equals(reversed)) {
            System.out.println("The text is a palindrome.");
        } else {
            System.out.println("The text is not a palindrome.");
        }
        scanner.close();
    }

    static String clean(String text) {
        String lower = text.toLowerCase(Locale.ROOT);
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < lower.length(); i++) {
            char c = lower.charAt(i);
            if (Character.isLetterOrDigit(c)) sb.append(c);
        }
        return sb.toString();
    }
}
```

`nextLine()` reads the whole line. Had `next()` been used here, only the first word, "A", would be read, because `next()` stops at a space. Wherever a sentence may be entered, `nextLine()` is needed.

`new StringBuilder(cleanText).reverse().toString()` does three jobs in a chain. First a modifiable copy of the text is created, then this copy is reversed in place, and finally it is converted back to a `String`. In Java, `String` objects are immutable; reversing a text, concatenating it or changing its letters always produces a new object. `StringBuilder`, on the other hand, is a modifiable sequence of characters and is designed for such jobs.

The `clean` method converts the text to lowercase, then walks over the characters one by one. The `charAt(i)` method gives the character at position `i`; positions start at 0 and end at `length() - 1`. `Character.isLetterOrDigit(c)` says whether a character is a letter or a digit. Letters of any alphabet, Turkish letters included, count as letters; spaces, commas and apostrophes do not. The suitable characters are appended to a `StringBuilder` and the result is returned.

Comparing text: `equals` and `==`. The most important line of the program is `if (cleanText.equals(reversed))`. If this line is written as `cleanText == reversed`, the program compiles and runs but says "not a palindrome" even for "abcba". The reason is what `==` means for objects. With primitive types such as `int`, `double` and `char`, `==` compares values. `String`,

however, is an object, and a variable of type `String` holds not the object itself but its place in memory, that is, a *reference* to the object. `==` checks whether two references point to the same object. `reversed` is a separate object freshly produced by `StringBuilder`; even if its content is the same as `cleanText`, it is another object and `==` gives `false`. Content comparison is done with the `equals` method.

This mistake also shows up in a sneaky way. Two identical texts written directly in the source code, `String a = "abc"; String b = "abc";`, are stored in Java as a single object, and `a == b` gives `true`. A programmer sees `==` working in a small experiment and gains a false confidence. For text read from the user or produced at run time, `==` gives `false` almost every time. In an experiment on JDK 21, `true` was obtained for two literal strings and `false` for "abcba" and the reverse produced by `StringBuilder`. The rule is simple: texts are always compared with `equals`.

Common mistake

Lowercasing text without specifying a locale. When the `toLowerCase()` method is called without a parameter, it uses the computer's locale. The capital "I" becomes "ı" in a Turkish locale and "i" in an English one: `"TITLE".toLowerCase()` gives "tıtle" on a computer set to Turkish. The same program can give different results on different computers. In the code above, `toLowerCase(Locale.ROOT)` is written to fix locale-independent rules. For Turkish text, the Turkish rules are selected with `Locale.forLanguageTag("tr")`; with that call "Ilık" becomes "ılık", while with `Locale.ROOT` the same word would become "ilık".

Tests. The program was run on JDK 21 with four tests: for "abcba", "Kayak" and "A man, a plan, a canal: Panama" it prints "The text is a palindrome."; for "Hello" it prints "The text is not a palindrome."

1.3.5.2 Prime check

Definition 1.3.6

A **prime number** is a natural number greater than 1 that is divisible only by 1 and itself: 2, 3, 5, 7, 11, 13, … Numbers greater than 1 that are not prime are called **composite numbers**.

The definition explicitly leaves two edge cases out: 0 and 1 are not prime. 2 is both the smallest prime and the only even prime. The test cases include these cases and ordinary numbers, both prime and composite (Table 1.3.19).

Input	Expected	Case tested
0, 1	not prime	Edge: the loop never runs
2	prime	Edge: the smallest prime, the loop never runs
7	prime	The loop runs to the end

Input	Expected	Case tested
21	not prime	A divisor is found early
25	not prime	The square of a prime

Table 1.3.19: Test cases for the prime check.

```
Start
    // Enter a number
    Read num
    // Flag: 0 and 1 are not prime
    If num < 2 then isPrime = False
    Else isPrime = True
    // Try divisors from 2 to half the number
    For divisor = 2 to num / 2
        If num % divisor = 0 then
            // If divisible, not prime
            isPrime = False
            Break
        End If
    End For
    // Print the result based on the flag
    If isPrime = True then
        Print num, " is a prime number."
    Else
        Print num, " is not a prime number."
    End If
End
```

The algorithm rests on three ideas.

The flag. The `isPrime` variable is a **flag**: a variable that takes only the value true or false and records whether a condition has occurred. At the start the number is assumed to be prime and the flag is raised. If the loop finds a divisor, the flag is lowered. When the loop ends, the state of the flag tells the result. The flag pattern is the general solution to questions of the type "is it true for all?": we start with an optimistic assumption, and a single counterexample refutes it.

Remainder and early exit. The condition `num % divisor = 0` tests whether the division leaves no remainder: since 21 % 3 = 0, 3 divides 21 exactly. As soon as a divisor is found it is certain that the number is not prime; trying the remaining divisors is pointless. `Break` (`break` in Java) leaves the loop at once.

The loop bound. Divisors are tried from 2 to `num / 2`. Apart from the number itself, no divisor can be larger than half the number. If a divisor *d* is larger than *num / 2*, the quotient *num / d* is less than 2, that is, 1; and this means *d* = *num*. `num / 2` is integer division: 7 / 2 = 3. For 0 and 1 the bound is 0; the loop never runs and the result is decided by the `If num < 2` decision in the first line. For 2 and 3 the bound is 1; the loop again does not run, the flag stays true and both numbers are correctly found prime.

Hand trace. Table 1.3.20 gives the trace for 7 and 21.

Number	Step	`divisor`	Remainder	`isPrime`
7 (bound 7 / 2 = 3)	7 ≥ 2, flag raised	—	—	True
	iteration 1	2	7 % 2 = 1	True
	iteration 2	3	7 % 3 = 1	True
	exit: 4 ≤ 3 false	4	—	True
	"7 is a prime number."			
21 (bound 21 / 2 = 10)	21 ≥ 2, flag raised	—	—	True
	iteration 1	2	21 % 2 = 1	True
	iteration 2	3	21 % 3 = 0	False
	Break	3	4 … 10 not tried	False
	"21 is not a prime number."			

Table 1.3.20: Trace of the prime algorithm for 7 and 21.

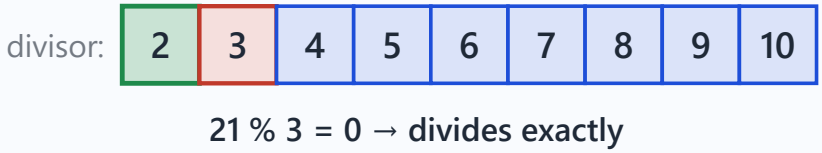


Figure 1.3.10: The divisors that could be tried for 21 are 2 … 10. 2 does not divide, 3 does; the loop is broken and 4 … 10 are never tried.

Java code.

```
import java.util.Scanner;

public class IsPrime {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int num = scanner.nextInt();
        boolean isPrime = true;
        if (num < 2) isPrime = false;
        for (int divisor = 2; divisor <= num / 2; divisor++) {
            if (num % divisor == 0) {
                isPrime = false;
                break;
            }
        }
        if (isPrime) {
            System.out.println(num + " is a prime number.");
        } else {
            System.out.println(num + " is not a prime number.");
        }
        scanner.close();
    }
}
```

In Java the flag is a variable of type `boolean`, and its values are `true` and `false`. The pseudocode line `If isPrime = True then` is written in Java as `if (isPrime)`. Since `isPrime` is already a `boolean`, there is no need to compare it with `true`. `if (isPrime == true)` also works correctly but is needlessly long. `if (isPrime = true)`, however, is an assignment: it compiles, makes `isPrime` `true` and always gives the result "prime". With `boolean` variables, the single-equals mistake slips past the compiler.

The `break` statement ends the nearest enclosing loop. Program flow moves to the first line after the loop, here the line `if (isPrime)`. Had `break` not been used, the result would not change; once the flag is lowered, it is not raised again. But the loop would needlessly try seven more divisors, from 4 to 10.

How many divisors are tried? The numbers of divisors the program tries were measured on JDK 21 with a copy that has a counter. For 7, 2 divisors (2, 3) are tried and the number is found prime. For 21, 2 divisors are tried; the loop is broken at 3. For 25, 4 divisors (2, 3, 4, 5) are tried. For 49, 6 divisors are tried and a divisor is found at 7. 97 is prime; the loop tries all 47 divisors from 2 to 48. Primes are the worst case of the algorithm, because the early exit never happens.

A better bound. The bound can be taken as $\sqrt{num}$ instead of `num / 2`. If a number splits into two factors $a \times b$, one of the factors is no larger than $\sqrt{num}$. So if no divisor is found up to $\sqrt{num}$, the number is prime. For 97 this bound is 9, and 8 divisors are tried instead of 47. In Java the condition is

written `divisor * divisor <= num`; this way there is no need to take a square root or use decimal numbers. With large numbers the difference is striking: for a prime around one billion, the `num / 2` bound means about 500 million trials, and the $\sqrt{num}$ bound about 31,600.

Common mistake

Forgetting the edge cases. If the line `if (num < 2) isPrime = false;` is removed, the loop never runs for 0 and 1, the flag stays `true` and the program declares both numbers prime. 0! in factorial, *n = 1* in Fibonacci, and 0 and 1 in the prime check must be tested separately. These inputs are the cases where the algorithm never enters the loop, and the correct result depends entirely on the initial values.

1.3.6 Search algorithms

1.3.6.1 The search problem

Searching asks whether a particular item is in a collection and, if so, where. It is one of the basic operations of almost every program that processes data. The place of a book in a library, a particular page on a website, a record in a database, a name in a phone book are all found by a search. To define the problem abstractly, imagine a basket and an apple.

Definition 1.3.7

The search problem. The input is a collection of objects (the "Basket") and a particular object (the "Apple"). The output is `true` if the apple is in the basket and `false` if not. In programs, instead, the **index** of the item found, that is, its position in the array, is usually returned; if the item is not found, −1 is returned.

Returning an index gives more information than returning `true` / `false`: both the presence and the position of the item are reported with a single number. In Java arrays the valid indices run from 0 to `length - 1`; since −1 can never be a valid index, it is safely used to mean "not found". Three search methods are examined in these notes (Figure 1.3.11). Random search looks at a random item at each step. Linear search looks at the items in order from start to end. Binary search looks at the middle item and eliminates half of the search space at each step.

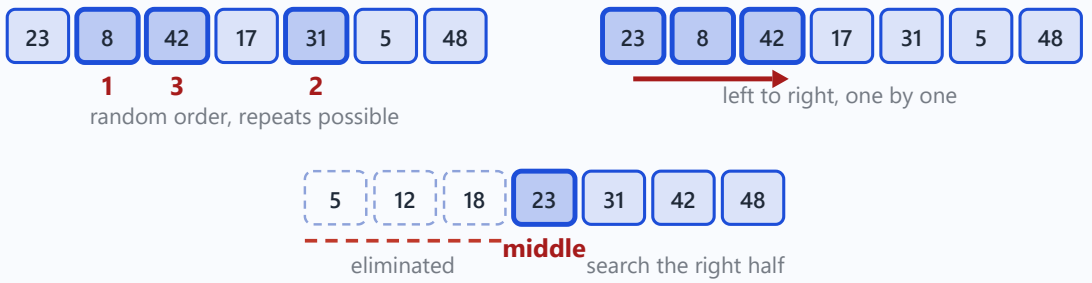


Figure 1.3.11: Three search methods. Random search: no order needed, no guarantee of finding. Linear search: no order needed, at most n comparisons. Binary search: the array must be sorted, at most about $log_2 n$ comparisons.

The measure for comparing the methods is the **number of comparisons**: how many times the sought item is compared with an element of the array. This number is independent of the machine, the language and the compiler. If it is known how many comparisons a method makes on a million items, it can be estimated how much faster it will be than the others on whatever computer it runs.

1.3.6.2 Random search

Random search checks the items of a collection in a random order. At each step a random item is chosen and compared with the target. If it matches, the search ends; if not, a new random item is chosen. The pseudocode is four steps:

1. Pick a random item.
2. Compare the chosen item with the target.
3. If the target is found, stop.
4. If not found, repeat the steps.

Figure 1.3.12 shows the search for 48 in an unsorted array of nine items. The randomly chosen items are 23, 42, 7 and 48 in turn; the search ends on the fourth try. If the same search were run again, other items would be chosen and the number of tries would change too.

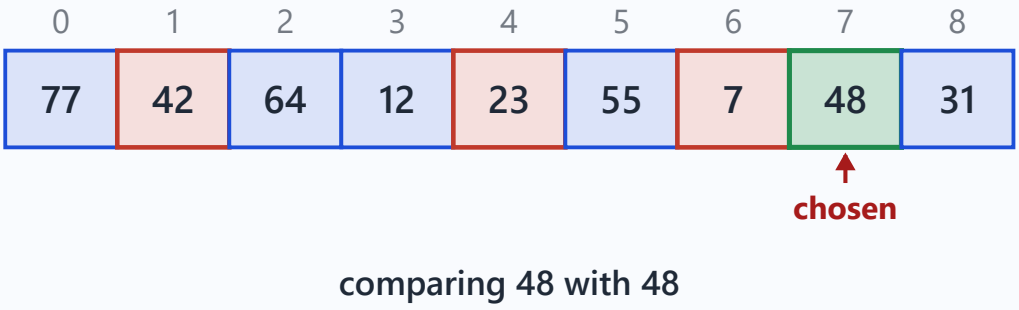


Figure 1.3.12: Random search, target 48. The items with red borders (23, 42, 7) are those chosen earlier that did not match; the fourth choice is 48.

Java code. The following method produces random indices with the `java.util.Random` class. For the method to compile, the line `import java.util.Random;` must be at the top of the file.

```
static int randomSearch(int[] array, int target, int maxTries) {
    Random random = new Random();
    for (int attempt = 1; attempt <= maxTries; attempt++) {
        int i = random.nextInt(array.length);   // 0 … length − 1
        if (array[i] == target) {
            return i;
        }
    }
    return -1;   // limit reached; no such item, or just bad luck
}
```

`random.nextInt(array.length)` produces a random integer between 0 and `array.length - 1`, both ends included; this range is exactly the valid indices of the array. The method's third parameter is `maxTries`. The biggest weakness of random search is that it does not stop if the target is not in the array: every attempt fails and the loop runs forever. This breaks the finiteness property of an algorithm. A try limit solves this problem but brings a new uncertainty. When the limit is reached, −1 is returned. It cannot be known whether this −1 means "the item is not there" or "the item is there but we were unlucky".

How many tries are needed? If there are n items in the array and a single target, the success probability of each try is $1/n$. Since the tries are independent, the average number of tries is n. The search for 48 in the nine-item array was repeated 200,000 times on JDK 21 and an average of 9.02 tries was measured. The probability of not finding it at all in n tries is $(1 - 1/n)^n$. For $n = 9$ this value is about 0.346: one search in three is still without a result after as many tries as there are items. Since the same item can be chosen again and again, even more tries than the worst case of linear search may be needed.

The advantages of random search are its simplicity and its independence of the order of the data. Its disadvantages are serious: there is no guarantee of a definite result, it may choose the same item many times, and if the target is absent it can be stopped only with a try limit. In practice this method is hardly used on its own as a search tool. Its value lies elsewhere: when the search space is too large to try exhaustively, for example in an optimization problem, random sampling is a useful starting point.

1.3.6.3 Linear search

Linear search is the most basic way of searching for an item in a collection. Starting from the first, the items are compared with the target in order. If a match is found the position is returned. If the end of the list has been reached, the item is not there. The items need not be sorted.

1. Put the items into an array.
2. Starting from the first item, compare in order with the target item.

3. If the target is found, stop and return its position.
4. If not found, move on to the next item.
5. Continue to the end of the list; if the end is reached, the item is not there.

```
static int linearSearch(int[] array, int target) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == target) {
            return i;
        }
    }
    return -1;
}
```

The method tries each index of the array in turn. When `array[i] == target` is true, `return i;` ends both the method and the loop at once; there is no need for `break`. If the loop ends without finding a match, flow reaches the last line and −1 is returned. The condition `i < array.length` stops at `length - 1`, the last valid index of the array.

Trace. When searching for 23 in the array [42, 17, 8, 23, 31], 42, 17 and 8 do not match; at the fourth comparison `array[3] = 23` is found and 3 is returned (Figure 1.3.13). Table 1.3.21 gives the numbers of comparisons for other targets in the same array. The values were verified by running a copy with a counter on JDK 21.



Figure 1.3.13: Linear search, target 23. On the left the first comparison (`i = 0` , 42); on the right the match at the fourth comparison (`i = 3`).

Target	Items looked at	Comparisons	Value returned
42	42	1 (best case)	0

Target	Items looked at	Comparisons	Value returned
23	42, 17, 8, 23	4	3
31	42, 17, 8, 23, 31	5	4
50	42, 17, 8, 23, 31	5 (worst case)	−1

Table 1.3.21: Linear search in the array [42, 17, 8, 23, 31].

Cost. In an array of n items, the best case is that the target is the first item: 1 comparison. The worst case is that the target is the last item or is not there at all: n comparisons. If the target is in the array and is equally likely to be at any position, the average is $(1 + 2 + \ldots + n) / n = (n + 1) / 2$ comparisons. The work grows in direct proportion to n: if the array doubles, the worst-case work doubles too. The algorithm gets its name from this linear growth.

Linear search is simple and easy to understand, works on unsorted data and gives a definite result: if the item exists, it is certainly found. It is the best choice for small collections. Its disadvantage is that it stays slow on large collections. In an array of a million items, a million comparisons are needed for an item that is not there.

1.3.6.4 Binary search

Binary search is a fast and efficient way of finding an item in sorted data. The idea is like looking up a word in a dictionary. The dictionary is opened in the middle; if the word comes before the opened page, we move to the left half, and if after, to the right half. At each step the number of pages to look at is halved.

1. Pick the middle item and compare it with the target.
2. If the target is found, stop.
3. If the target is smaller than the middle item, focus on the left half.
4. If the target is larger than the middle item, focus on the right half.
5. Repeat until the target is found or the search space is exhausted.

The third and fourth steps rest on the array being sorted. If the target is smaller than the middle item and the array is sorted from smallest to largest, every item in the right half is greater than or equal to the middle one; the target cannot be there. This inference holds only in a sorted array.

```
static int binarySearch(int[] array, int target) {
    int low = 0, high = array.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (array[mid] == target) {
            return mid;
        } else if (target < array[mid]) {
            high = mid - 1;
        } else {
            low = mid + 1;
        }
    }
    return -1;
}
```

The search space is the region between the indices `low` and `high`, both ends included. At the start it is the whole array. In each iteration `mid = (low + high) / 2` is computed. This is integer division; if the number of items in the space is even, it picks the left of the two middle items. If the target equals the middle item, the index is returned. If it is smaller, `high = mid - 1` eliminates the right half and the middle item. If it is larger, `low = mid + 1` eliminates the left half and the middle item. In both cases the middle item is not examined again; it has already been compared with the target. The loop condition is `low <= high`. When `low` passes `high`, the search space is empty and the item is not there.

Trace. Let 48 be searched for in the sorted array [7, 12, 23, 31, 42, 48, 55, 64, 77]. Table 1.3.22 shows the variables in each iteration, and Figure 1.3.14 three moments.

Iteration	`low`	`high`	`mid`	`array[mid]`	Decision
1	0	8	(0 + 8) / 2 = 4	42	48 > 42 → `low = 5`
2	5	8	(5 + 8) / 2 = 6	55	48 < 55 → `high = 5`
3	5	5	(5 + 5) / 2 = 5	48	equal → `return 5`

Table 1.3.22: Binary search, target 48 in the array [7, 12, 23, 31, 42, 48, 55, 64, 77].

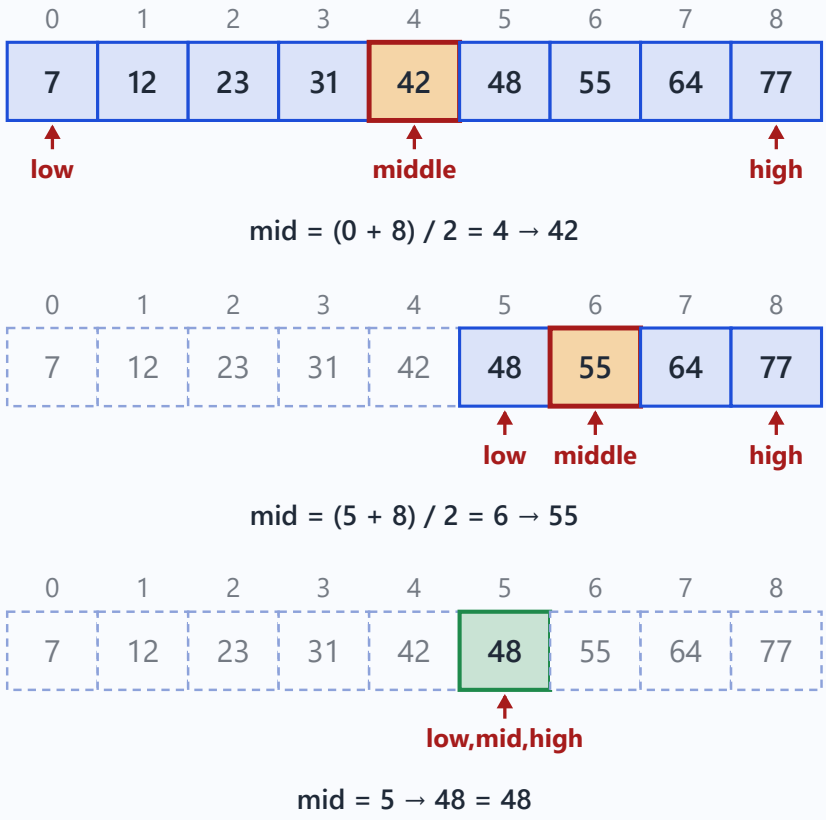


Figure 1.3.14: The three iterations of binary search. Items with dashed borders have been eliminated. In the second iteration the search space has four items (48, 55, 64, 77); the left of the two middle ones, 55, is chosen. In the nine-item array only 3 comparisons were made.

Example 1.3.1

Question. If 50 is searched for in the same array, which items are looked at and what does the method return?

Solution. The first two iterations are the same as in the search for 48: `mid = 4` (42) and since 50 > 42, `low = 5`; `mid = 6` (55) and since 50 < 55, `high = 5`. In the third iteration `mid = 5`, `array[5] = 48`, and since 50 > 48, `low = 6`. Now `low = 6`, `high = 5`; `low <= high` is false and the loop ends. The method returns −1 after 3 comparisons. In the final state `low` shows the position where 50 would have to be in the array: between 48 and 55, index 6.

In the same array, two comparisons suffice for 12 (42 → 12). Four comparisons are needed for 77 (42 → 55 → 64 → 77). In this nine-item array no search exceeds four comparisons. These numbers were verified by running a copy with a counter on JDK 21.

Binary search on an unsorted array. The dangerous side of binary search is that on an unsorted array it gives a wrong result without any error. Let 77 be searched for in the array [77, 42, 64, 12, 23, 55, 7, 48, 31]. The first middle item is 23; since 77 > 23, we move to the right half. Then 7, 48 and 31

are looked at; since 77 is larger than all of them, we keep going right. In the end the space is empty and −1 is returned. Yet 77 is the first item of the array. The method does not check whether the array is sorted and gives no warning. When each of the nine items of this array is searched for separately, only two (23 and 48) are found. And those found happen to fall at the middle positions by chance. On unsorted data the result is left to chance; that is why it is more accurate to say "may give a wrong result on an unsorted array".

How many comparisons? Each iteration halves the search space. A space of *n* items falls to at most *n/2* after one iteration and at most *n/4* after two. About $log_2\ n$ iterations pass until the space falls to a single item; the last item is also looked at once. The worst-case number of comparisons is $\lfloor log_2\ n \rfloor + 1$. For 1,000,000 items, $log_2\ 1{,}000{,}000 \approx 19.93$; taking the floor and adding one gives 20 comparisons. Linear search can make a million comparisons on the same array.

> **Common mistake**
>
> Off by one at the bounds. If the loop condition is written `low < high`, the loop stops when a single item is left in the space and that item is never looked at: when searching for 48 in [7, 12, …, 77], the third iteration never runs and −1 is returned. If `high = mid` or `low = mid` is written, the space does not shrink in some cases and the loop runs forever. The right pattern is this: the condition `low <= high`, the updates `mid - 1` and `mid + 1`.

Java's built-in binary search. The `Arrays.binarySearch(array, target)` method in the `java.util.Arrays` class implements the same algorithm. If the item is found, it returns its index. If not found, instead of −1 it returns a negative number of the form `-(insertionPoint) - 1`. The insertion point is the index at which the item would enter the array without breaking the order. For 50 in [7, 12, …, 77] the insertion point is 6, so the method returns −7. A negative result means "not found". This method too assumes the array is sorted; its documentation states explicitly that the result is undefined for an unsorted array.

1.3.6.5 Comparing linear and binary search

The difference between the two methods is insignificant on small arrays and striking on large ones (Table 1.3.23).

Number of items *n*	Linear (*n*)	Binary ($\lfloor log_2\ n \rfloor + 1$)
9	9	4
100	100	7
1.000	1.000	10
1.000.000	1.000.000	20
2.000.000	2.000.000	21

Number of items n	Linear (n)	Binary ($\lfloor \log_2 n \rfloor + 1$)
1.000.000.000	1.000.000.000	30

Table 1.3.23: Number of comparisons in the worst case. The binary search column was computed with the formula $\lfloor \log_2 n \rfloor + 1$ and was also verified by trying all targets for $n \le 2{,}000{,}000$.

The most instructive pair of rows in the table is 1,000,000 and 2,000,000. When the array doubles, the work of linear search doubles too. The work of binary search, however, goes up by only one, because the extra iteration brings the doubled space back to its old size.

This advantage has a price: the data must be sorted, and sorting has a cost too. For a single search, first sorting an array and then binary searching is more expensive than simply searching linearly. Binary search pays off if many searches will be made on the same data, or if the data is kept sorted anyway. That is why phone books and dictionaries are in alphabetical order: they are sorted once and searched millions of times.

	Random	Linear	Binary
Needs sorted data?	No	No	Yes
Finds the item if it is there?	No guarantee	Yes	Yes (on sorted data)
Stops if the item is absent?	Only with a try limit	Yes, in n comparisons	Yes, in $\lfloor \log_2 n \rfloor + 1$ comparisons
Average comparisons (item present)	n	$(n + 1)/2$	about $\log_2 n$

Table 1.3.24: Comparison of the three search methods.

1.3.7 Sorting algorithms

1.3.7.1 The sorting problem

Sorting arranges the items of a collection by a certain rule. For numbers the rule is usually smallest to largest (ascending) or largest to smallest (descending); for text it is alphabetical order. Sorting is a basic preliminary step of data analysis and of searching. In a sorted array the smallest and largest items are at the ends, the median is in the middle, and repeated items sit side by side. Most importantly, binary search works only on a sorted array.

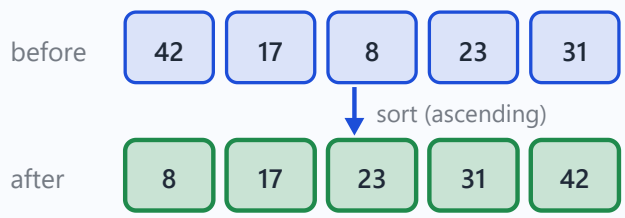


Figure 1.3.15: Ascending sort: the array [42, 17, 8, 23, 31] becomes [8, 17, 23, 31, 42].

Three simple methods are examined in these notes. **Random sort** (shuffling) puts the items in a random order and does not guarantee sorting; it is the opposite job of sorting but uses the same swap operation. **Selection sort** picks the smallest item and puts it at the front. **Bubble sort** compares neighboring items and carries the larger one toward the end. There are faster methods too: merge sort and quicksort. Java's built-in `Arrays.sort` method uses improved forms of these.

The common building block of the three methods is the **swap**: exchanging two items of the array. In Java there is no single operation that directly exchanges the values of two variables; a swap is done in three steps with a temporary variable.

```
int temp = array[i];
array[i] = array[j];
array[j] = temp;
```

The first line copies the value of `array[i]` to a safe place. The second line writes `array[j]` over `array[i]`; the old value of `array[i]` is now only in `temp`. The third line puts this value into `array[j]`.

> **Common mistake**
>
> Swapping without a temporary variable. The lines `a = b; b = a;` make both variables the value of `b`. If we start with `a = 3`, `b = 5`, after the first line `a = 5` and 3 is lost. The second line puts 5 into `b` again. The experiment run on JDK 21 prints "5 5"; the correct swap with a temporary variable prints "5 3".

1.3.7.2 Random sort (shuffling)

Random sort, or **shuffling**, puts the items of a collection in a random order. Shuffling a deck of cards, changing the order of exam questions for each student, and producing random test data for a program are typical uses. Some algorithms also avoid bad cases by shuffling the input first. Quicksort, when its input has been shuffled beforehand, is not affected by specially crafted bad inputs.

Figure 1.3.16 shows the first and last state of shuffling the array [42, 17, 8, 23, 31] three times in a row. The first shuffle produced [31, 17, 42, 8, 23], the second [8, 23, 31, 42, 17], the third [8, 23, 17, 31, 42]. The last array has come close to order but is still not sorted. Shuffling does not guarantee sorting; that is not its purpose either.

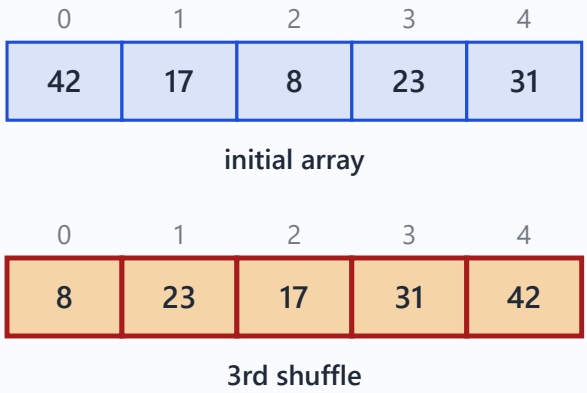


Figure 1.3.16: Shuffling. On the left the initial array, on the right the result of the third shuffle: [8, 23, 17, 31, 42]. Each shuffle produces a different result.

A correct shuffle. For a shuffle to count as well done, every arrangement must come up with equal probability. *n* items have *n!* different arrangements; each must have probability *1/n!*. The classic method that satisfies this condition is the **Fisher–Yates shuffle**. We start from the end of the array. For each position `i`, a random `j` is chosen from 0 to `i`, both ends included, and `array[i]` and `array[j]` are swapped. Then we move one position to the left.

```
static void shuffle(int[] array) {
    Random random = new Random();
    for (int i = array.length - 1; i > 0; i--) {
        int j = random.nextInt(i + 1);   // 0 … i
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}
```

The item that will settle in the last position is chosen with equal probability from each of the *n* items. The second-to-last position gets one of the remaining *n − 1* items, and so on. The total number of choices is *n × (n − 1) × … × 2 = n!*, and each sequence of choices corresponds to a different arrangement. The method makes exactly *n − 1* swaps. The code was run 600,000 times on JDK 21 with the array [1, 2, 3]. Each of the six arrangements was observed between 99,651 and 100,509 times; the expected value is 100,000. In Java the same job is done by the ready-made `Collections.shuffle(list)` method, which also uses the Fisher–Yates method.

Common mistake

Choosing `j` from the whole array at every step. A loop that chooses with `random.nextInt(array.length)` also shuffles the array, but the arrangements do not come out with equal probabilities. This loop produces n^n equally likely sequences of choices. Since this number is not divisible by *n!*, some arrangements come up more often than others. For

three items, $3^3 = 27$ sequences of choices cannot be divided evenly among six arrangements: for the array [1, 2, 3], the arrangements [1, 3, 2], [2, 1, 3] and [2, 3, 1] come up with probability 5/27 and the others with 4/27. In 270,000 trials on JDK 21 the first three arrangements were observed about 50,000 times each and the others about 40,000 times each.

Sorting by shuffling. A sorting method can be devised that says "shuffle until the array is sorted". This method is called bogosort and is hopelessly slow. Of the 10! = 3,628,800 different arrangements of 10 items, only one is sorted. The probability that a shuffle produces the sorted array is about one in three and a half million. On average 3.6 million shuffles are needed, each followed by a check for sortedness. For 20 items this number rises to 2.4×10^{18}.

1.3.7.3 Selection sort

Selection sort is a simple algorithm that sorts by comparing items. The smallest item in the unsorted part of the array is found and moved to the start of that part. Then the unsorted part becomes one item shorter and the process is repeated.

1. Find the smallest item in the collection.
2. Move this item to the start of the collection (swap it with the first item).
3. Find the smallest item in the remaining part.
4. Move it to the second position.
5. Repeat these steps to the end of the collection.

```
static void selectionSort(int[] array) {
    int n = array.length;
    for (int i = 0; i < n - 1; i++) {
        int smallest = i;
        for (int j = i + 1; j < n; j++) {
            if (array[j] < array[smallest]) smallest = j;
        }
        int temp = array[i];
        array[i] = array[smallest];
        array[smallest] = temp;
    }
}
```

The `i` variable of the outer loop is the position where the smallest item will settle. The items to the left of `i` are sorted and in their final places; `i` and everything to its right is not yet sorted. The inner loop walks from `i + 1` to the end of the array and keeps the index of the smallest item seen so far in `smallest`. This is the classic pattern for finding the smallest item in an array: the first item is taken as the candidate, and the candidate is replaced whenever a smaller item is seen. When the inner loop ends, `array[i]` and `array[smallest]` are swapped. The outer loop stops at `n - 1`, because when the last item is left alone it is already in place.

Step by step. Let the array [64, 25, 12, 22, 11, 36, 48] be sorted. Table 1.3.25 shows each outer iteration, and Figure 1.3.17 the situation before and after the first step and the final state.

Step (`i`)	Unsorted part	Smallest (`smallest`)	Swap	Comp.	Array after the step
1 (0)	index 0–6	11 (4)	64 ↔ 11	6	[11, 25, 12, 22, 64, 36, 48]
2 (1)	index 1–6	12 (2)	25 ↔ 12	5	[11, 12, 25, 22, 64, 36, 48]
3 (2)	index 2–6	22 (3)	25 ↔ 22	4	[11, 12, 22, 25, 64, 36, 48]
4 (3)	index 3–6	25 (3)	none (already in place)	3	[11, 12, 22, 25, 64, 36, 48]
5 (4)	index 4–6	36 (5)	64 ↔ 36	2	[11, 12, 22, 25, 36, 64, 48]
6 (5)	index 5–6	48 (6)	64 ↔ 48	1	[11, 12, 22, 25, 36, 48, 64]

Table 1.3.25: Selection sort, [64, 25, 12, 22, 11, 36, 48]. "Comp." is the number of comparisons of the inner loop in that iteration.

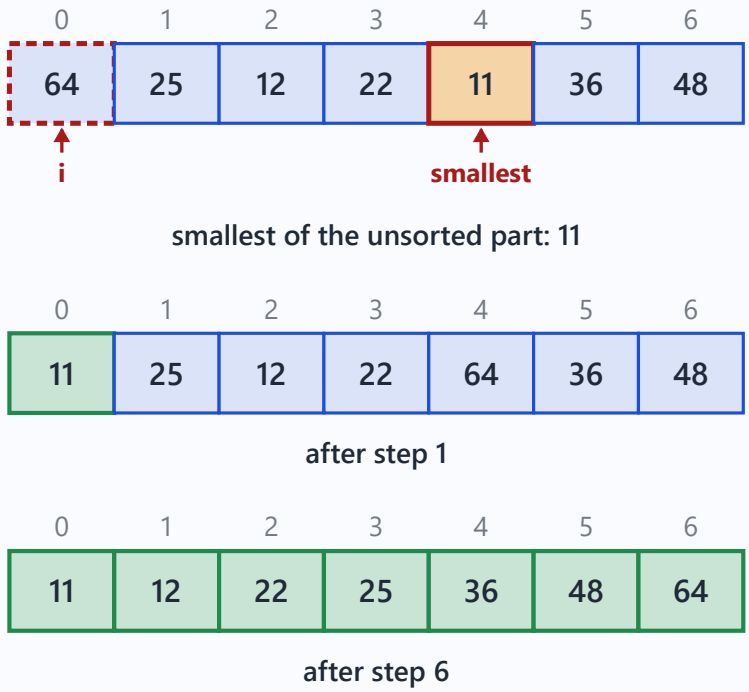


Figure 1.3.17: Selection sort. (a) Step 1: the smallest of the unsorted part, 11 (index 4), is about to be swapped with the 64 at position 0. (b) After step 1, 11 is in place (green). (c) After step 6 the array is sorted; the last item has settled into place by itself.

In the fourth step the smallest item is already at the start of the unsorted part; `smallest` and `i` are equal. In this case too the code runs the swap lines: the item is swapped with itself and the array does not change. So the number of swaps the code makes is always $n - 1$; in this example 6. The number of swaps that actually move something is 5. If desired, an `if (smallest != i)` check can be added before the swap.

Cost. The inner loop makes $n - 1$ comparisons in the first iteration, $n - 2$ in the second, and 1 in the last. The total number of comparisons is

$$(n - 1) + (n - 2) + \ldots + 2 + 1 = n(n - 1)/2$$

For 7 items, $7 \cdot 6 / 2 = 21$; the sum of the comparisons in the table is also 6 + 5 + 4 + 3 + 2 + 1 = 21. This number does not depend on the initial order of the array. Even if the array is already sorted, selection sort makes 21 comparisons, because to find the smallest it must look at every remaining item. There is no way for the algorithm to notice that the array is sorted. The number of swaps is at most $n - 1$. This is the strength of selection sort: in situations where writing is expensive, for example in a memory that wears out with each write, making few swaps is an advantage.

Example 1.3.2

Question. When the array [29, 10, 14, 37, 13] is sorted with selection sort, what does the array look like after each step? How many comparisons and how many real swaps are made?

Solution. Step 1: the smallest is 10 (index 1); it is swapped with 29 → [10, 29, 14, 37, 13]. Step 2: among indices 1–4 the smallest is 13 (index 4); it is swapped with 29 → [10, 13, 14, 37, 29]. Step 3: among indices 2–4 the smallest is 14 and it is already in place → [10, 13, 14, 37, 29]. Step 4: the smaller of 37 and 29 is 29; they are swapped → [10, 13, 14, 29, 37]. The number of comparisons is 4 + 3 + 2 + 1 = 10 = $5 \cdot 4 / 2$. The number of real swaps is 3; the number of swaps the code makes, counting the swap with itself, is 4.

1.3.7.4 Bubble sort

Bubble sort compares items standing side by side. If the left item is larger than the right one, the two are swapped. One such pass from the start of the array to the end is called a **pass**. During a pass the largest item is carried step by step to the right, like a bubble rising to the surface of water, and settles in the last position of the array at the end of the pass.

1. Start at the beginning of the collection.
2. Compare two items side by side; if needed, swap them and carry the larger one to the right.
3. Repeat this step to the end of the collection (one pass).
4. Since the largest item has arrived at the end, regard the last item as sorted.
5. Repeat the first four steps for the remaining part; if no swap happens in a pass, stop.

```
static void bubbleSort(int[] array) {
    int n = array.length;
    for (int pass = 0; pass < n - 1; pass++) {
        boolean swapped = false;
        for (int j = 0; j < n - 1 - pass; j++) {
            if (array[j] > array[j + 1]) {
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}
```

The `pass` variable of the outer loop is the number of completed passes. After each pass one more item at the end of the array settles into its final place. That is why the inner loop, with the condition `j < n - 1 - pass`, stops one item earlier on each pass; there is no point in comparing items that are already in place again. The reason `n - 1` is used in the bound of the inner loop is the access `array[j + 1]`. `j` can be at most `n - 2`; otherwise `j + 1` runs outside the array.

The `swapped` variable is again a flag. It is lowered at the start of each pass and raised when a swap is made. If the flag is still down at the end of a pass, no swap happened in that pass. This shows that every neighboring pair is already in the right order, that is, that the array is sorted. `break` skips the remaining passes. This **early stopping** feature makes bubble sort fast on sorted or nearly sorted arrays.

Step by step. Let the same array as in selection sort, [64, 25, 12, 22, 11, 36, 48], be sorted with bubble sort. Table 1.3.26 shows the result of each pass, and Figure 1.3.18 the first and last comparison of the first pass and the final state.

Iteration	Steps	Comp.	Swap	Array after the pass	Settled in place
1	1–6	6	6	[25, 12, 22, 11, 36, 48, 64]	64
2	7–11	5	3	[12, 22, 11, 25, 36, 48, 64]	48
3	12–15	4	1	[12, 11, 22, 25, 36, 48, 64]	36
4	16–18	3	1	[11, 12, 22, 25, 36, 48, 64]	25
5	19–20	2	0	[11, 12, 22, 25, 36, 48, 64]	no swap → stop
Total		20	11		

Table 1.3.26: Bubble sort, [64, 25, 12, 22, 11, 36, 48]. The step numbers in the slides count each comparison as one step.

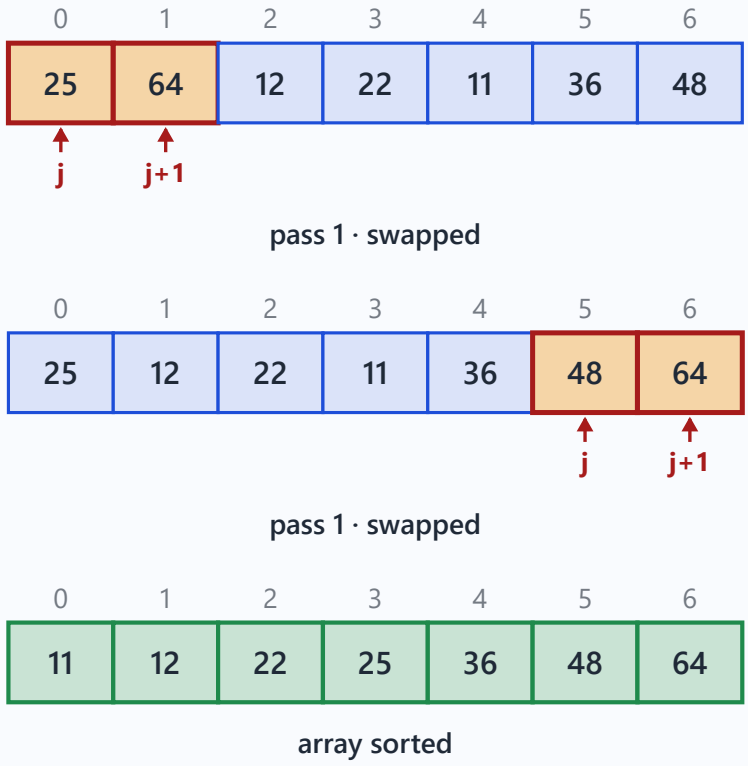


Figure 1.3.18: Bubble sort. (a) Step 1: 64 > 25, they were swapped. (b) Step 6: 64 has reached the last position. (c) In pass 5 no swap happened; the array is sorted.

In the first pass 64 wins every comparison and travels to the very end with six swaps. In the second pass 25 first passes 12 and 22, then 11; it gets stuck at 36. From then on the bubble is 36, but since 36 < 48 no swap happens; 48 happens by chance to be in its own place already at the end of this pass. The third and fourth passes carry 11 to the left step by step: in bubble sort, small items can move left only one position per pass. Large items near the start of the array go to the end quickly, while small items near the end come to the front slowly. In the fifth pass no swap happens and the algorithm stops. In total 20 comparisons and 11 swaps were made. The values were verified by running a copy with a counter on JDK 21.

Number of swaps and inversions. The number of swaps in bubble sort has a special meaning. Every **inversion** in the array, that is, a pair made of a larger item on the left and a smaller item on the right, is fixed by exactly one swap. Swapping two neighboring items removes exactly one inversion. So the number of swaps equals the number of inversions in the array. In the array [64, 25, 12, 22, 11, 36, 48], there are 6 items to the right of 64 that are smaller than it; to the right of 25 there are 3 (12, 22, 11), to the right of 12 there is 1 (11), and to the right of 22 there is 1 (11). In total 6 + 3 + 1 + 1 = 11 inversions and 11 swaps.

Cost. The worst case is that the array is in reverse order. In the array [64, 48, 36, 25, 22, 12, 11] every comparison ends in a swap and early stopping never kicks in. 6 + 5 + 4 + 3 + 2 + 1 = 21 comparisons and 21 swaps are made; in general $n(n - 1)/2$ comparisons and the same number of swaps. The best

case is that the array is already sorted. In the first pass $n - 1$ comparisons are made, no swap happens and the algorithm stops. For 7 items, 6 comparisons suffice. Selection sort still makes 21 comparisons on the same sorted array.

Example 1.3.3

Question. Bubble sort is applied to the array [5, 1, 4, 2, 8]. What does the array look like after the first pass? With early stopping, how many passes, comparisons and swaps are made?

Solution. Pass 1: 5 > 1 swap → [1, 5, 4, 2, 8]; 5 > 4 swap → [1, 4, 5, 2, 8]; 5 > 2 swap → [1, 4, 2, 5, 8]; 5 < 8 no swap. After the pass the array is [1, 4, 2, 5, 8] (4 comparisons, 3 swaps). Pass 2: 1 < 4; 4 > 2 swap → [1, 2, 4, 5, 8]; 4 < 5 (3 comparisons, 1 swap). Pass 3: 1 < 2, 2 < 4; no swap and the algorithm stops (2 comparisons). In total 3 passes, 9 comparisons and 4 swaps. The inversions in the array are (5, 1), (5, 4), (5, 2) and (4, 2); their number is also 4.

Common mistake

Writing the inner loop with `j < n`. At the last step of the inner loop `j = n - 1` and `array[j + 1]`, that is, `array[n]`, is tried to be read. The program stops with a runtime error such as `ArrayIndexOutOfBoundsException: Index 7 out of bounds for length 7`. In every loop that accesses `array[j + 1]`, the upper bound is taken one less.

1.3.7.5 Comparing the sorting methods

The properties of the three methods are side by side in Table 1.3.27. The last row gives the values measured for the same seven-item array.

	Random (shuffle)	Selection	Bubble
Idea	Produce a random arrangement	Pick the smallest, put it at the front	Compare neighbors, carry the larger to the end
Is the result sorted?	No guarantee	Yes	Yes
Comparisons (n items)	—	always $n(n - 1)/2$	worst $n(n - 1)/2$, $n - 1$ on a sorted array
Swaps	$n - 1$	at most $n - 1$	worst $n(n - 1)/2$ (as many as the number of inversions)
Example array ($n = 7$)	—	21 comparisons, 5 real swaps	20 comparisons, 11 swaps

Table 1.3.27: Shuffling, selection sort and bubble sort.

The work of selection sort and bubble sort grows with n^2. For 1,000 items, $1000 \cdot 999 / 2 = 499{,}500$, about 500 thousand comparisons are made. For 1,000,000 items this number rises to about 500 billion. Even a computer that makes a billion comparisons a second would spend more than eight minutes on this job. On big data, smarter algorithms are used. The work of methods such as merge sort grows with $n \cdot \log_2 n$; for a million items that means about 20 million comparisons. The difference is a factor of about 25,000.

Determine	Idea	Average	Worst	Note
Bubble	Neighboring items are compared and swapped if needed	$O(n^2)$	$O(n^2)$	Simple; ineffective on big data
Selection	The smallest is found and placed at the front; repeat with the rest	$O(n^2)$	$O(n^2)$	Slow on big data; few swaps
Insertion	The next item is inserted at the right place in the sorted part	$O(n^2)$	$O(n^2)$	Effective on small or nearly sorted data
Merge	Split in two, sort each half, merge	$O(n \log n)$	$O(n \log n)$	Effective on big data; needs extra memory
Quick	A pivot is chosen; smaller items to the left, larger to the right	$O(n \log n)$	$O(n^2)$	Very fast on average

Table 1.3.28: Comparison of five sorting algorithms. The $O(...)$ notation gives how the work grows with n, dropping constant factors.

Two notations are used in this table: $O(n^2)$ and $O(n \log n)$. "Big O" notation roughly describes how an algorithm's work grows with the size of its input. Instead of exact numbers, the shape of the growth is given. An algorithm that makes $n(n - 1)/2$ comparisons is $O(n^2)$. As n grows, the term that determines the value of this expression is $n^2/2$; the factor 1/2 does not change the shape of the growth. Linear search is $O(n)$ and binary search is $O(\log n)$. This notation is used as a common language for comparing algorithms.

In Java there is no need to rewrite these algorithms every time to sort an array. The call `Arrays.sort(array)` sorts arrays of primitive types such as `int` with an improved form of quicksort, and arrays of objects with a method based on merge sort. The purpose of learning the simple algorithms is not to use them in production. The aim is to acquire the habits of looping, comparing, swapping and counting, and to be able to work out the cost of an algorithm on your own.

1.3.8 Common mistakes

The most frequent mistakes in the algorithms of this chapter fall into two groups. The first group is mistakes in the algorithm itself: they appear whatever the language is and are found by hand tracing (Table 1.3.29). The second group is specific to Java: it shows up as a message from the compiler or the JVM, and sometimes with no message at all (Table 1.3.30).

Mistake	Result	The fix
Initializing the accumulator wrongly: `factorial = 0`	Every product is 0; the result is always 0	Start at 0 for addition and at 1 for multiplication
Off by one in the loop bound: `i = 2 … n` in Fibonacci	*n + 1* terms are printed	Trace the bounds by hand with a small test
Using binary search on an unsorted array	−1 for 77 in [77, 42, 64, 12, 23, 55, 7, 48, 31]; no error message	Sort first or use linear search
Forgetting edge cases	0 and 1 come out prime in the prime check	Test 0, 1, empty input, *n = 1* separately
No stopping condition in random search	If the target is absent the loop runs forever	Put in a try limit
Swapping without a temporary variable: `a = b; b = a;`	Both variables get the value of `b`	Save `temp = a` first
Writing the shift order backwards in Fibonacci	0 1 1 2 4 8 … (a doubling sequence)	First `a = b`, then `b = result`
Strict `<` in the three-number comparisons	7 5 5 for 5 5 7	Use `<=`

Table 1.3.29: Common mistakes at the algorithm level.

Mistake	What happens?
`import java.util.Scanner;` forgotten	Compile error: `error: cannot find symbol … symbol: class Scanner`
No `;` at the end of the line: `int total = 0`	Compile error: `error: ';' expected`
Single `=` in a comparison: `if (num % divisor = 0)`	Compile error: `error: unexpected type … required: variable, found: value`
Single `=` with `boolean`: `if (isPrime = true)`	Compiles; the condition is always true
Comparing strings with `==`	Compiles but gives `false` for "abcba" and its reverse; `equals` must be used
Loop with `i <= array.length`	Runtime error: `ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5`
Using `int` for 13!	No error message, wrong result: 1932053504 (correct 6227020800); `long` must be used
Entering "3.5" on a machine set to Turkish	Runtime error: `java.util.InputMismatchException`
Writing `^` for exponentiation: `2 ^ 10`	Compiles; the result is 8 (bitwise exclusive or)

Table 1.3.30: Common Java-specific mistakes. The messages were produced on JDK 21.

A significant share of the mistakes in the tables produce no message at all. Overflow, comparing strings with `==`, binary search on an unsorted array and the `boolean` assignment mistake do not stop the program; they only corrupt the result. The only safeguard against these mistakes is the test cases, prepared when the problem was defined and including the edge cases. With compile errors, on the other hand, the message usually points at the place of the mistake correctly; messages are read starting from the first, because a single missing piece often brings several more messages after it.

1.3.9 Summary

- **The problem-solving cycle:** define the problem, determine the algorithm, translate to code, test; if there is a bug, go back. Test cases are prepared before the code is written and include edge cases (0, an empty string, equal numbers, $n = 1$).
- **Algorithm:** a sequence of instructions made of clear steps in a definite order and finite in number. Its five properties are input, output, definiteness, finiteness and effectiveness.
- **Pseudocode and hand tracing:** pseudocode is a notation written for people, independent of language rules. A trace table shows the variable values at each step and catches loop-bound mistakes without writing code.
- **Translating to Java:** `Read` → `nextInt` / `nextDouble` / `nextLine` with `Scanner`; `Print` → `print` / `println`; equality `==`, assignment `=`; "and" `&&`, "or" `||`; `Break` → `break`. Between two `int`s, `/` is integer division.
- **Decision algorithms:** sorting two numbers needs a single `if … else`. With three numbers, the smallest is found first and the remaining two are sorted; `<=` is needed to handle equalities correctly; all 3! = 6 orders are tested.
- **Loop algorithms:** factorial and exponentiation work with an accumulator starting at 1; Fibonacci keeps the last two terms and shifts them each iteration. `int` is enough up to 12!, F(46) and 2^{30}; beyond that it overflows silently. `long` is enough up to 20! and F(92).
- **Palindrome and prime:** the text is first cleaned, reversed and compared with `equals`. The prime check uses a flag and `break`; the bound is `num / 2`, better `divisor * divisor <= num`.
- **Searching:** random search makes n tries on average and gives no guarantee. Linear search makes n comparisons in the worst case and works on unsorted data. Binary search needs sorted data and makes $\lfloor \log_2 n \rfloor + 1$ comparisons in the worst case: 20 for 1,000,000 items.
- **Sorting:** a swap uses a temporary variable. The Fisher–Yates shuffle produces every arrangement with equal probability. Selection sort always makes $n(n - 1)/2$ comparisons and at most $n - 1$ swaps. Bubble sort makes $n(n - 1)/2$ comparisons in the worst case; its number of swaps equals the number of inversions, and with early stopping it finishes in $n - 1$ comparisons on a sorted array. Both are $O(n^2)$.

1.3.10 Key terms

algorithm pseudocode trace table test case edge case dry run accumulator flag overflow compiler bytecode runtime error integer division palindrome prime number linear search binary search random search shuffle selection sort bubble sort swap inversion

1.3.11 Test Yourself

1. The factorial program is run with `num = 6`. How many iterations does the loop make and what is the result? What would be printed if the loop condition were mistakenly written `i < num`?

   Answer

   The loop makes 6 iterations for `i` = 1 … 6; `factorial` becomes 1, 2, 6, 24, 120, 720 in turn and 720 is printed. With `i < num` the last factor 6 is skipped and 5! = 120 is printed.

2. What does the Fibonacci program print for *n = 10*, and how many iterations does the loop make? If the positions of the lines `a = b;` and `b = result;` in the loop body are swapped, what is the output for *n = 6*?

   Answer

   0 1 1 2 3 5 8 13 21 34; the loop makes *n − 2 = 8* iterations for `i` = 2 … 9. If the lines are swapped, `b = result` comes first, then `a = b` gives `a` the new term as well, and the two variables become equal: 0 1 1 2 4 8.

3. How many divisors does the prime program try for 49? If the loop condition is changed to `divisor * divisor <= num`, how many divisors are tried for 97?

   Answer

   For 49 the bound is 49 / 2 = 24; 2, 3, 4, 5, 6 do not divide, 7 does and the loop is broken: 6 tries. With the new condition, for 97 the divisors with $d^2 \le 97$ are 2 … 9; none divides and 97 is found prime after 8 tries. The old condition made 47 tries.

4. What do these three lines print: `System.out.println("total " + 2 + 3);`, `System.out.println(2 + 3 + " total");`, `System.out.println(7 / 2 * 2);`?

   Answer

   "total 23", "5 total" and "6". In the first line the string comes first, so both `+` are concatenation. In the second, 2 + 3 = 5 is added first. In the third, 7 / 2 becomes 3 by integer division, and 3 * 2 = 6.

5. In the sorted array {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}, which items does binary search look at for

23 and for 4, and what does it return?

Answer

23: `mid` = 4 (16) → right, `mid` = 7 (56) → left, `mid` = 5 (23) → found; 3 comparisons, returned value 5. 4: 16 → left, `mid` = 1 (5) → left, `mid` = 0 (2) → right; `low` = 1, `high` = 0, and after 3 comparisons −1 is returned.

6. The array [9, 4, 7, 1, 3] is sorted with selection sort. What does the array look like after each step? How many comparisons and how many real swaps are made?

Answer

[1, 4, 7, 9, 3] → [1, 3, 7, 9, 4] → [1, 3, 4, 9, 7] → [1, 3, 4, 7, 9]. Comparisons 4 + 3 + 2 + 1 = 10; since in each step the smallest item is at another position, 4 real swaps.

7. How many passes, comparisons and swaps does bubble sort (with early stopping) make on the arrays [2, 3, 4, 5, 1] and [5, 1, 2, 3, 4]? Although the two arrays have the same number of inversions, why do the numbers of passes differ?

Answer

[2, 3, 4, 5, 1]: 4 passes, 4 + 3 + 2 + 1 = 10 comparisons, 4 swaps. [5, 1, 2, 3, 4]: 2 passes, 4 + 3 = 7 comparisons, 4 swaps. Both arrays have 4 inversions, so the numbers of swaps are equal. But while a large item can travel all the way to the end in one pass, a small item moves only one position left per pass. Bringing the 1 at the end to the front takes four passes.

8. In a sorted array, how many comparisons do linear and binary search make in the worst case for 2,000,000 items? What were the numbers when there were 1,000,000 items?

Answer

Linear search 2,000,000; binary search $\lfloor \log_2 2{,}000{,}000 \rfloor + 1 = 20 + 1 = 21$. For 1,000,000 items the numbers were 1,000,000 and 20. When the array doubles, the work of linear search doubles and that of binary search goes up by one.

9. In the exponentiation program, if the variable `result` had been declared `int` instead of `long`, what would be printed for 2^{31} and 2^{32}? Why is there no error message?

Answer

−2147483648 and 0. `int` holds at most $2^{31} - 1 = 2{,}147{,}483{,}647$; one more wraps around to the smallest `int` value, and one more multiplication turns it into 0. Java does not check integer overflow and throws no exception; the result is silently wrong.

1.3.12 Further reading

- A. B. Downey and C. Mayfield, *Think Java* (2nd ed., 2019): the chapters on conditionals, loops, strings and arrays complement the Java details of these notes in plain language. The book can be read free online.
- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Sections 1.1 and 2.1: treats binary search and the basic sorting algorithms with Java code and experimental measurements.
- P. Deitel and H. Deitel, *Java: How to Program (Early Objects)* (11th ed., 2017), Chapters 4 and 19: explains algorithm development, pseudocode and search and sort algorithms with many complete programs.
- D. E. Knuth, *The Art of Computer Programming, Vol. 2* (3rd ed., 1997), Section 3.4.2: contains the classic treatment of the Fisher–Yates shuffle and of random permutations.

1.3.13 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
2. P. Deitel, H. Deitel, *Java: How to Program (Early Objects)*, 11th ed., Pearson, 2017 (Chapter 4: Algorithms and pseudocode; Chapter 19: Searching and sorting).
3. A. B. Downey, C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019.
4. D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3rd ed., Addison-Wesley, 1997.
5. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed., Addison-Wesley, 1998.
6. Oracle, *The Java Tutorials: Language Basics*, docs.oracle.com/javase/tutorial.
7. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011 (Section 1.1: binary search; 2.1: elementary sorts).

PART I · CHAPTER 1.4

Flowcharts

Before a program is written, the solution has to be planned on paper. A flowchart draws this plan with boxes and arrows. These notes introduce the symbols of flowcharts and the rules for drawing them, and show the Java counterpart of each symbol. Then come the three kinds of flow that are the building blocks of all algorithms: sequential, conditional and repetitive flow. Each kind is explained with a diagram, a Java program written from that diagram, and a step-by-step trace of the program. The counter, accumulator, flag and sentinel patterns often used in loops, early exit and nested loops are treated separately. The last sections cover the subprogram, that is, the method in Java, tracing a diagram at a desk, and the choice of test cases. All the Java programs in the examples were compiled and run with JDK 21; the values in the trace tables were checked against these runs.

LEARNING OUTCOMES

Concept
Explain the relationship between an algorithm and a flowchart; say what a flowchart is good for and where it falls short.

Symbols
Recognize the Start/End, input, process, decision, loop, print, connector and subprogram symbols; choose the right symbol for each step; apply the drawing rules.

Control structures
Tell sequential, conditional and repetitive flow apart; draw a diagram for each; tell a structured diagram from an unstructured one.

Tracing
Execute a diagram or a program step by step on a given input by building a trace table; notice boundary errors and infinite loops.

From diagram to Java
Translate each symbol into its Java counterpart: `nextInt()`, assignment, `if` / `else`, `while`, `for`, `println`, a method call; tell the errors the compiler catches from those it cannot.

Design and testing
Plan a simple problem first with a diagram and then turn it into a program; choose test cases that exercise every path and the boundary values.

1.4.1 Algorithm and flowchart

A Java program consists of statements that the computer executes in order. But starting to write a program directly at the keyboard often gives a faulty result. First you think about how to solve the problem and fix the steps of the solution; only then are these steps translated into the programming language. The flowchart is one of the oldest and most widespread tools for the stage of fixing the steps.

1.4.1.1 From algorithm to drawing

Definition 1.4.1

An **algorithm** is an ordered and finite plan of the steps to follow to solve a problem or carry out a task. Every step is clear; with the same input the same result is reached every time.

Definition 1.4.2

A **flowchart** is a drawing that shows visually the steps of an algorithm, the connections between the steps and the control mechanisms. Each step is drawn with a symbol. The order in which the steps run is shown by arrows, that is, by **flow lines**.

The relationship between the two definitions can be explained with a recipe. A recipe is an algorithm: prepare the ingredients, boil the water, brew the tea. The same recipe drawn with boxes and arrows is a flowchart. It is also possible to describe the recipe in writing. But a decision like "if the water has not boiled, wait; if it has, brew" or a repetition like "wait until it boils" is easily overlooked in text. In a drawing, the decision stands out at once as a diamond and the repetition as an arrow looping back. Figure 1.4.1 is the diagram of this recipe.

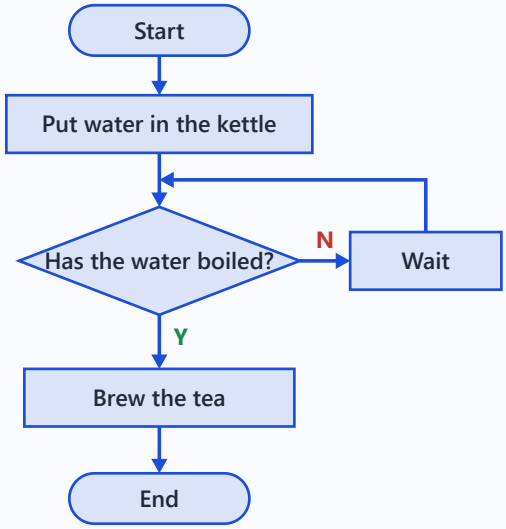


Figure 1.4.1: The flowchart of the tea-brewing algorithm. The No (N) exit of the decision box goes to the "Wait" box; the arrow leaving it returns to the top of the decision box. When the water boils, the Yes (Y) exit leads on to the "Brew the tea" step.

The diagram is read from top to bottom. Execution begins at the round-ended *Start* box and proceeds along the arrows. "Put water in the kettle" is a process and is drawn with a rectangle. "Has the water boiled?" is a question; since its answer is Yes or No, it is drawn with a diamond and has two exits. The No exit goes to the "Wait" box and from there back to the question itself. So the question is asked again and again until the answer is Yes. This small drawing contains all three kinds of flow: steps one after another, a decision and a repetition.

A flowchart is not tied to a programming language. A Java, C or Python program can be written from the same diagram. So the diagram is not the program itself but the plan of the program. The correctness of the plan is checked on paper; only after that is code written.

1.4.1.2 Benefits and limits

The most important benefit of a flowchart is that it makes a complex process visible at a glance. Which step follows which, and in which situation a step is skipped or repeated, are plainly visible with arrows. This visibility lets mistakes be found before code is written. For example, if only the Yes exit of a decision box has been drawn, it is immediately clear that nobody has thought about what happens in the No case.

The second benefit of the diagram is documentation. A process, for example a registration procedure or the operating steps of a machine, is described with a flowchart in reports and technical documents. The third benefit is that it provides a common language. Not everyone in a team may know the same programming language, but everyone can read a flowchart.

The diagram has limits too. The diagram of a large program spills over several pages and becomes hard to read. When the algorithm changes, the diagram has to be redrawn, which takes time. As detail increases, so does the number of symbols. In that case **pseudocode**, which writes the algorithm in semi-natural language, is more practical. In practice flowcharts are used for small and medium-sized algorithms, or to show the outline of a large program.

The development of a program usually goes in the order of Figure 1.4.2. The problem is defined, the solution steps, that is, the algorithm, are determined, the algorithm is put into a flowchart, the diagram is translated into a Java program, and the program is run and tested. If testing gives a wrong result, we go back and fix the algorithm.

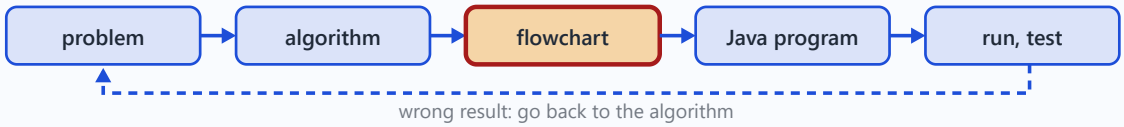


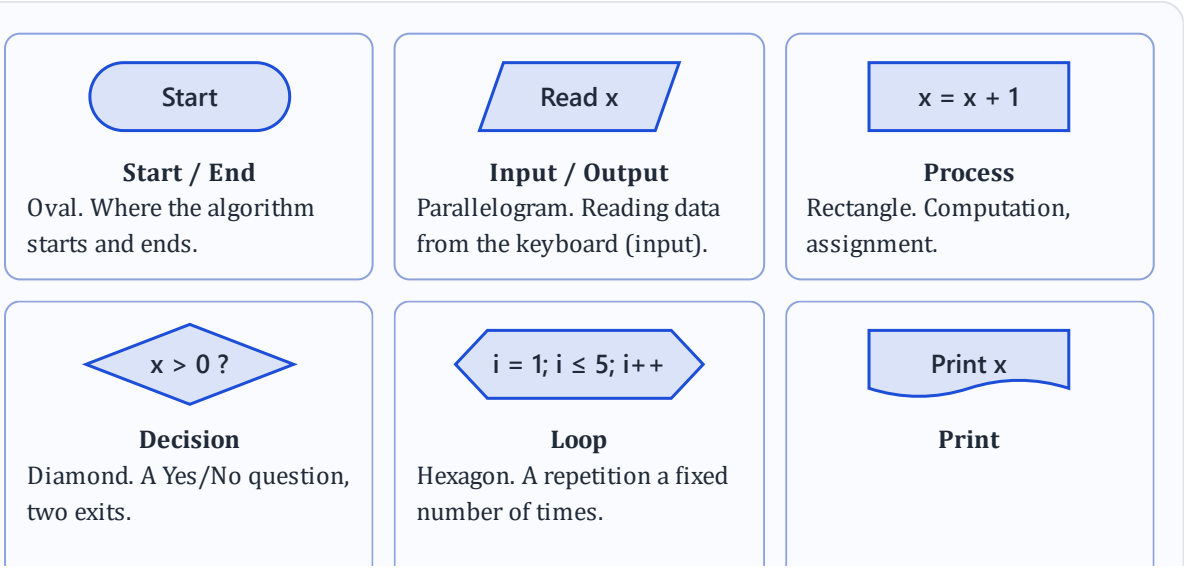
Figure 1.4.2: The typical working order: problem → algorithm → flowchart → Java program → running and testing. If testing gives a wrong result, the dashed arrow goes back to the algorithm.

1.4.2 Symbols and drawing rules

The shape of each symbol in a flowchart tells the kind of that step. Even before reading the text inside a box, the reader can tell from its shape whether a step is a read, a computation or a decision. So choosing the symbols correctly is the first condition for a diagram to be understandable.

1.4.2.1 Basic symbols

The shapes of the symbols are based on an international standard, ISO 5807. Figure 1.4.3 shows the eight most commonly used symbols.



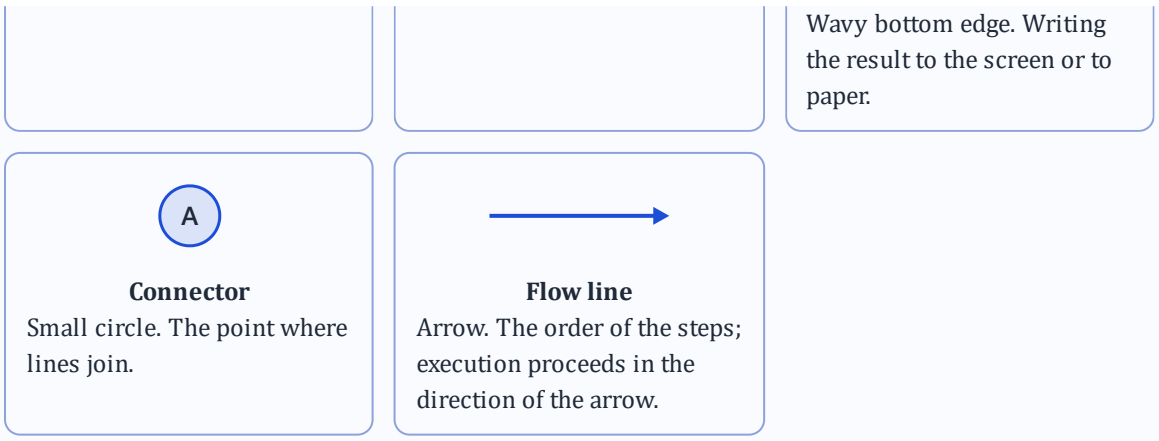


Figure 1.4.3: The basic flowchart symbols and their roles.

The **Start/End** symbol is a rectangle with rounded sides, that is, an oval. Every diagram has a Start symbol; execution begins there. The End symbol is where the algorithm finishes.

The **Input** symbol is a parallelogram and shows a step where data is taken from outside. A parallelogram labeled "Read num" means the user will enter a number at the keyboard and this number will be put into the variable named *num*. Some sources draw output, that is, writing to the screen, with a parallelogram too. In these notes a separate symbol is used for writing to the screen, the **Print** symbol with a wavy bottom edge. The wavy edge recalls a sheet of paper; the old name of the symbol is "document".

The **Process** symbol is a plain rectangle. A computation or an assignment is written inside it: "area = a × b", "counter = counter + 1". A process box has a single entry and a single exit.

The **Decision** symbol is a diamond. A question whose answer is Yes or No is written inside: "num > 0 ?". The decision box is the only symbol with two exits. One of the exits is labeled Yes (Y), the other No (N).

The **Loop** symbol is an elongated hexagon. It summarizes a job repeated a fixed number of times in a single box: "i = 1; i ≤ 10; i++". The three parts inside the hexagon are the initialization, the condition and the update. The **Connector** is a small circle. It shows where several arrows join; for example, when the two branches of a decision finish their work, they meet at the connector. The **flow line** is the arrow that connects the symbols. Execution always proceeds in the direction the arrow points.

1.4.2.2 Java counterparts of the symbols

When a diagram is translated into a Java program, each symbol corresponds to a specific Java construct. Table 1.4.1 gives this mapping. The table can be used like a dictionary in going from diagram to code.

Symbol	Text in the diagram	Java counterpart
Start / End	Start … End	`public static void main(String[] args) { … }`

Symbol	Text in the diagram	Java counterpart
 Input	Read num	`int num = scanner.nextInt();`
 Process	result = num1 + num2	`int result = num1 + num2;`
 Decision	result = 0 ?	`if (result == 0) { … } else { … }`
 Decision + back arrow	counter ≤ 10 ?	`while (counter <= 10) { … }`
 Loop	i = 1; i ≤ 10; i++	`for (int i = 1; i <= 10; i++) { … }`
 Print	Print result	`System.out.println(result);`

Table 1.4.1: Flowchart symbols, the expression written in the box, and the Java counterpart.

The first row of the table says that everything between the Start and End symbols of the diagram is written in the body of the `main` method in Java. The program starts at the first line of the `main` method and ends when it reaches the last line. For the Input symbol, Java uses the `java.util.Scanner` class. At the start of the program, an object that reads from the keyboard is created with the line `Scanner scanner = new Scanner(System.in);`. Then for each "Read" box, `scanner.nextInt()` (integer), `scanner.nextDouble()` (decimal number) or `scanner.next()` (word) is called.

The decision symbol has two different Java counterparts. If there is no arrow going back, the decision is an `if` statement: the question is asked once, and one of the two branches runs depending on the answer. If an arrow leaving one of the branches of the decision returns to the top of the decision box, the structure is a loop and is written with `while`. So the same diamond can be an `if` or a `while`, depending on where the arrows go. The hexagonal loop symbol corresponds directly to the `for` statement.

The point where most mistakes are made in translation is the equals sign. In a process box in the diagram, "=" means **assignment**: the value on the right is computed and put into the variable on the left. In a decision box, "=" means **comparison**: the question is whether two values are equal. Java writes these two meanings with different signs: assignment with `=`, comparison with `==`. Similarly, "mod" (the remainder of a division) in the diagram is the `%` operator in Java; the signs ≤ and ≥ are written `<=` and `>=`, and ≠ is written `!=`.

Common mistake

The question "result = 0 ?" in the decision box is translated into Java as `if (result = 0)`. This line does not compile. The JDK 21 compiler gives this message: `error: incompatible types: int cannot be converted to boolean`. `result = 0` is an assignment and its value is the `int` 0; `if` expects a `boolean` condition. The right way is `if (result == 0)`. In C the same mistake compiles and silently runs wrongly; Java's type checking catches it at compile time.

1.4.2.3 Other symbols

The basic symbols are enough for most diagrams. To split long diagrams or to show where data is written, ISO 5807 has other symbols too. Figure 1.4.4 shows six of them.

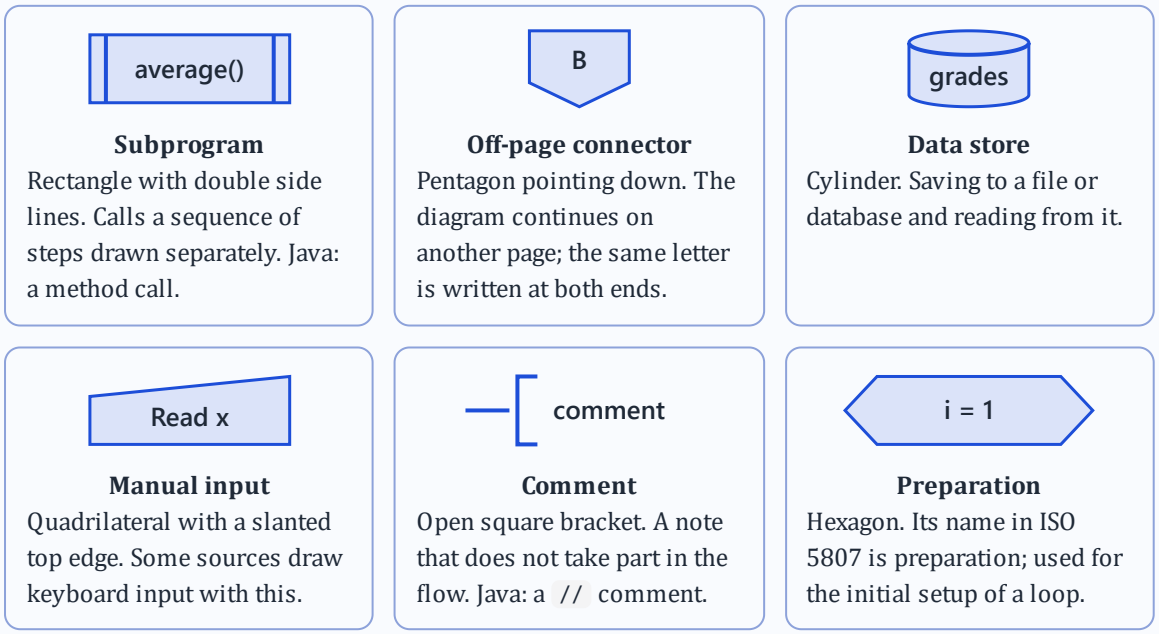


Figure 1.4.4: Additional flowchart symbols.

The **Subprogram** symbol is a rectangle with vertical lines added at its two sides. It calls a sequence of steps drawn elsewhere; its Java counterpart is a method call. One part of a long diagram is drawn separately and called with this symbol in the main diagram. The **off-page connector** is used when a diagram does not fit on a single page. A letter is written in the pentagon at the end of the first page, for example B. The pentagon at the start of the second page carries the same letter; the reader follows the flow by this letter.

The **Data store** symbol is a cylinder and shows data written to a lasting place, for example a file or a database. The **Manual input** symbol is used to emphasize keyboard input separately; many sources prefer the parallelogram instead. The **Comment** symbol is not part of the flow. It is for jotting a note next to a box; it resembles the `//` comment in Java. Just as the compiler ignores comments, a person following the diagram does not include comments in the execution.

The shape of the **Preparation** symbol is the same as the loop symbol. In ISO 5807 this hexagon is called "preparation" and shows the setup before a job begins. The use of this symbol to gather a loop's initial value, condition and update in one box comes from here.

1.4.2.4 Drawing rules

A flowchart is built in a few steps. First the operations are determined: which inputs will be read, which computations done, which outputs written? Then the steps are put in a logical order. Next all the symbols are connected to one another with flow lines. Finally, decision points are added for the steps that depend on a condition.

The following rules are observed to keep the drawing readable:

- A diagram has a single Start and at least one End. Two separate End symbols do not break the rule; but two Start symbols leave unclear where execution begins.
- Flow is read from top to bottom and left to right. Arrows that go outside this direction, for example the back arrow of a loop, are clearly shown with the arrowhead.
- Every symbol except a decision has a single exit. Only the decision symbol has two exits.
- Decision exits are always labeled Yes/No (Y/N). An unlabeled exit does not say which answer it belongs to.
- Lines do not cross one another. If a crossing is unavoidable, the lines are joined at a connector.
- Short and clear expressions are written in the boxes: "Read a, b", "area = a × b", "Print area". Long sentences make the diagram crowded.

1.4.2.5 Step-by-step drawing: the area of a rectangle

The rules can be applied to a small problem. The problem is to find the area of a rectangle whose side lengths are entered at the keyboard.

Example 1.4.1

Problem: print the area of a rectangle on the screen from the two side lengths read from the keyboard.

1. **Skeleton.** First a Start and an End symbol are drawn. All the steps will go between these two.
2. **Inputs.** Two side lengths are needed to compute the area. They are read from the keyboard: a parallelogram labeled "Read a, b" is added.
3. **Process.** The area is the product of the two sides. A rectangle labeled "area = a × b" is added. This box computes the product and assigns the result to the variable *area*.
4. **Output.** The computed area is written on the screen: a Print symbol labeled "Print area" is added.
5. **Connecting.** The symbols are connected with flow lines in a logical order: Start → Read → Process → Print → End. There is no decision point in this problem; the flow is sequential.

6. **Testing.** The diagram is tried with an input. If a = 4 and b = 3 are entered, area = 4 × 3 = 12 is computed and 12 is seen on the screen. The expected result is also 12; the diagram works correctly.

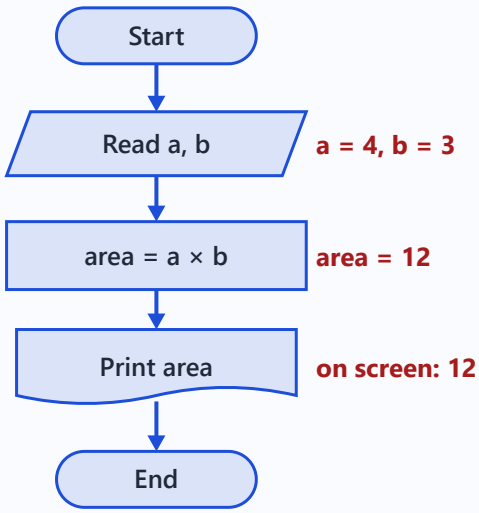


Figure 1.4.5: The diagram that finds the area of a rectangle. The red notes on the right show the values the variables take in the test with the input a = 4, b = 3.

Once the diagram is ready, translating it into Java is mechanical. With the help of Table 1.4.1 each symbol becomes a line:

```
import java.util.Scanner;

public class RectangleArea {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int a = scanner.nextInt();
        int b = scanner.nextInt();
        int area = a * b;
        System.out.println(area);
        scanner.close();
    }
}
```

The "Read a, b" in the parallelogram becomes two separate `nextInt()` calls. The process box turns into the line `int area = a * b;`; the × sign in the diagram is written `*` in Java. The Print symbol is the line `System.out.println(area);`. When the program is run with the input 4 and 3, it prints 12 on the screen; this is the same as the result in the diagram's test. Each box runs only once, no step is skipped and no step is returned to. These properties make up the definition of sequential flow.

1.4.3 Control structures

Which statement of a program runs in which order is called **control flow**. In a flowchart, the control flow is the arrows themselves. However tangled the arrows may look, every well-drawn diagram is built from only three basic structures.

1.4.3.1 The three kinds of flow

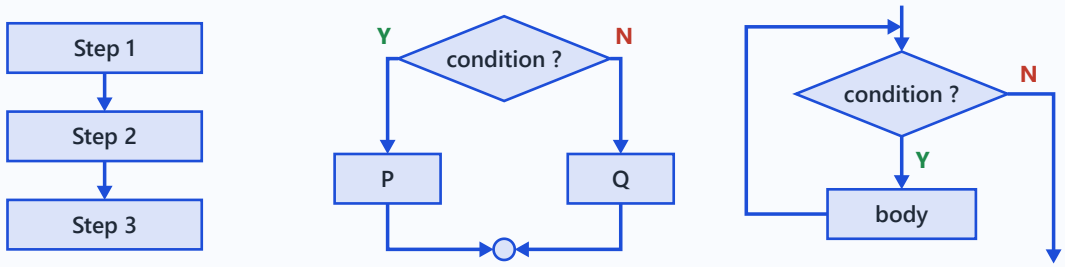


Figure 1.4.6: The three control structures. (a) Sequential flow: the steps run in order, once each. (b) Conditional flow: depending on the condition, P or Q runs; the paths join at the connector. (c) Repetitive flow: while the condition holds the body repeats; when the condition is false, execution continues from the N exit.

Definition 1.4.3

Sequential flow is flow in which the steps run in the order they are written and each runs exactly once. **Conditional flow** is flow in which one of two or more paths is chosen according to the result of a condition. **Repetitive flow**, or a **loop**, is flow in which some steps run again and again as long as a condition holds.

Figure 1.4.6 shows the three structures side by side. In sequential flow the arrows always go down; no arrow leaves or returns. In conditional flow two arrows leave a decision box; these two paths then join again at a connector. In repetitive flow, an arrow returns back to the top of the decision box. An arrow going back is the sure sign of a loop.

It has been shown mathematically that these three structures are sufficient. Corrado Böhm and Giuseppe Jacopini proved the following result in an article they published in 1966:

Theorem 1.4.1 (Structured program theorem, Böhm–Jacopini)

Every computable algorithm can be written using only three structures: sequence, selection and repetition. These structures are either arranged one after another or nested inside one another.

The counterpart of the theorem in Java is clear. Sequence is statements written one after another. Selection is `if` / `else`, and repetition is the `while` and `for` statements. Java's other control statements, for example `switch` or `do-while`, are shorthand for these three structures. The theorem guarantees that even a complex problem can be solved with these three pieces. The same idea is followed when designing a diagram: the problem is divided into sequence, selection and repetition pieces and the pieces are combined.

1.4.3.2 Structured and unstructured diagrams

Definition 1.4.4

A **structured flowchart** is a flowchart in which every control structure has a single entry and a single exit. The structures are either arranged one after another or nested inside one another; no arrow enters from outside into the middle of a structure.

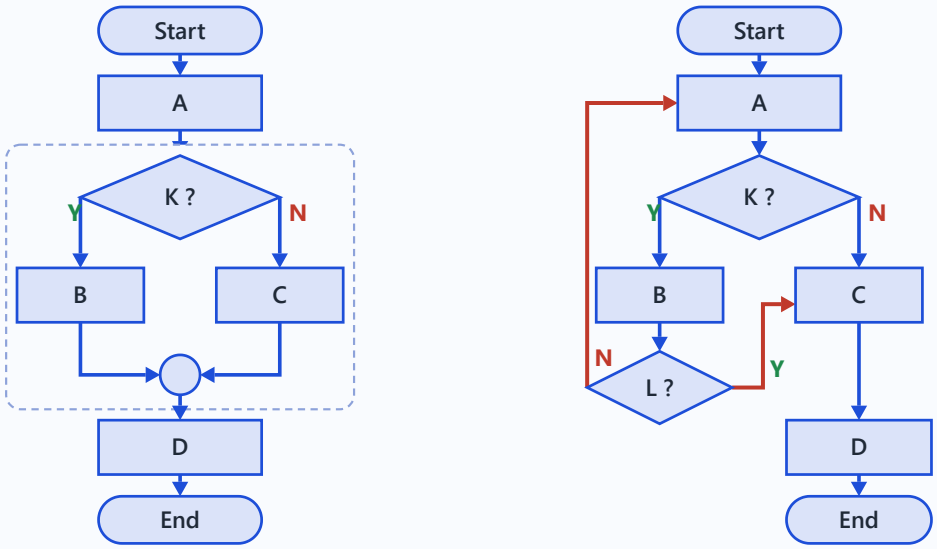


Figure 1.4.7: (a) A structured diagram: the decision K with the branches B and C forms a block with a single entry and a single exit inside the dashed frame. (b) An unstructured diagram: the Yes arrow of the decision L enters the middle of the other branch (C), and the No arrow goes back to A.

In the diagram on the left of Figure 1.4.7, the block that starts with the decision K is shown with a dashed frame. A single arrow enters the block from the top and a single arrow leaves from the bottom. The branches B and C stay inside the block and join at the connector. It is possible to think of this block as a single box without knowing what is inside. The person reading the diagram first sees A, then the block, then D.

In the diagram on the right, the Yes arrow of the decision L enters the middle of the other branch, the box C. The No arrow goes back to A. In this case box C can be reached by two different paths; A too is reached both from Start and from L. To tell which path a box was reached by, the whole diagram has

to be traced. The arrows tangle with one another like spaghetti; that is why such flow is called "spaghetti flow". Spaghetti flow is hard to follow and to find a bug in.

In old programming languages such jumps were written with the `goto` statement. The statement `goto label;` would take execution to any place in the program. In Java `goto` is a reserved word but has no meaning; it cannot be used. A program containing `goto start;` does not compile on JDK 21, with the message `error: illegal start of expression`. The designers of Java deliberately kept this word out of the language.

Java has two limited jumps: `break` and `continue`. `break` ends the loop it is in at once. `continue` skips the rest of that iteration of the loop and moves to the next iteration. These two statements take execution not to an arbitrary place in the program but only to the end or the start of the loop they are in. So no spaghetti flow arises.

A practical consequence of the Böhm–Jacopini theorem is this: every unstructured diagram can, if needed, be turned into a structured one by adding a `boolean` **flag** variable. The flag records whether an event has happened; this variable is tested in place of the jump.

1.4.4 Sequential flow

Sequential flow is the simplest of the three structures. Still, every program contains sequential parts: giving variables initial values, reading input, computing a formula, writing the result. Even the conditional and repetitive structures consist of sequential steps inside themselves.

1.4.4.1 Properties of sequential flow

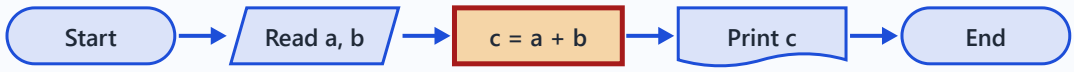


Figure 1.4.8: Sequential flow: Start → Read a, b → c = a + b → Print c → End. The flow proceeds without branching or returning.

Sequential flow is arranging the jobs to be done one after another in a logical order. Figure 1.4.8 draws such a flow from left to right; the same flow can also be drawn from top to bottom. Sequential flow has four properties.

It is simple. There is no branching or repetition. To understand the flow it is enough to follow the arrows in order.

It is easy to maintain. Since the steps run in the order they are written, it is easy to find where the program is when looking for a bug. Debugging is nothing but checking the steps one by one.

Order matters. If a step uses the result of the previous one, it cannot change places. The step "c = a + b" cannot be done before a and b are read. But the order of independent steps can be changed. If the places of "Read a" and "Read b" are swapped, the result does not change; only the order in which the user is asked changes.

There is no going back. Each step runs exactly once. No step is skipped and no earlier step is returned to. A sequential flow of N steps always ends in N steps, independent of the input.

Common mistake

If the addition step is written before the numbers are read, the addition is attempted with values not yet known. In the diagram this is a logic error. In Java, if variables are declared but have not yet received a value, the compiler catches the mistake:

```
int num1, num2;
int result = num1 + num2;     // the numbers have not been read yet
num1 = scanner.nextInt();
num2 = scanner.nextInt();
```

JDK 21 gives two errors for this fragment: `error: variable num1 might not have been initialized` and the same for `num2`. Java never allows a local variable that has not received a value to be read. This rule is called *definite assignment*. For each variable the compiler examines all paths of the program; if there is a path that reaches the point where the variable is used without it having received a value, it does not compile the program.

1.4.4.2 The sum of two numbers

The classic example of sequential flow is printing the sum of two numbers entered at the keyboard. The algorithm for the problem has six steps:

1. Start.
2. Read the first number (num1).
3. Read the second number (num2).
4. Add the two numbers and assign the result to the variable result.
5. Write the result on the screen.
6. End.

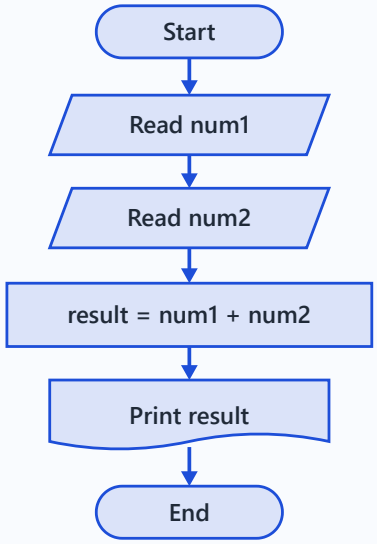


Figure 1.4.9: The flowchart that prints the sum of two numbers. Each step of the algorithm corresponds to one symbol.

Figure 1.4.9 is the diagram of this algorithm. Each step of the algorithm has become a symbol: Start and End are ovals, the two reading steps are parallelograms, the addition is a rectangle, and writing to the screen is the Print symbol. The algorithm has six steps and the diagram has six symbols. This one-to-one correspondence is a typical property of sequential flow.

The reading steps are drawn as two separate parallelograms. A single "Read num1, num2" box would also have been correct. Two separate boxes show clearly that in the Java program too there will be two separate reads and two separate prompt messages.

1.4.4.3 From diagram to Java: the sum program

The diagram is translated into the following Java program:

```
import java.util.Scanner;

public class AddTwoNumbers {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("First number: ");
        int num1 = scanner.nextInt();
        System.out.print("Second number: ");
        int num2 = scanner.nextInt();
        int result = num1 + num2;
        System.out.println("Result: " + result);
        scanner.close();
    }
}
```

The program runs line by line as follows. The `import` statement on line 1 brings the `Scanner` class from the `java.util` package into the program. Without this line the compiler does not recognize the name `Scanner`. Line 3 defines the class named `AddTwoNumbers`; in Java every program is written inside at least one class, and the file must be named `AddTwoNumbers.java`.

The `main` method on line 4 is the Start symbol in the diagram; the program begins here. On line 5 the `Scanner` object that will read from the keyboard is created. `System.in` is standard input, that is, the keyboard.

Lines 6 and 7 are the first parallelogram. `print` first tells the user what to enter. Unlike `println`, it adds no line end; so the number the user types appears on the same line as the message. `scanner.nextInt()` waits for the user to type an integer and press Enter. The value read is assigned to the variable `num1`. Lines 8 and 9 do the same job for the second number.

Line 10 is the process box: `int result = num1 + num2;`. Line 11 is the Print symbol. In the expression `"Result: " + result`, the `+` operator concatenates a string and a number; the number is converted to a string and appended. Line 12 closes the `Scanner` object. On line 13 `main` ends; this is the End symbol in the diagram.

Table 1.4.2 is the trace of the program when the user enters 7 first and then 5. In the table each row corresponds to a diagram box and the lines of code that make up that box. The "—" sign shows that the variable has not yet received a value.

Line	Diagram box	`num1`	`num2`	`result`	Added to the screen
4–5	Start; the `Scanner` is created	—	—	—	
6–7	Read num1	7	—	—	First number: 7
8–9	Read num2	7	5	—	Second number: 5
10	result = num1 + num2	7	5	12	
11	Print result	7	5	12	Result: 12
12–13	End	7	5	12	

Table 1.4.2: Trace of the `AddTwoNumbers` program with the input 7 and 5.

The conversation seen on screen is as follows; the numbers the user types come right after the messages:

```
First number: 7
Second number: 5
Result: 12
```

The trace table shows the properties of sequential flow with numbers. The table has six rows and each diagram box appears exactly once. No row has been skipped and no row has been returned to. The `result` variable receives a value only on line 10; before that, "—" is shown. Even if the input

changes, the number of rows in the table does not; only the values in the cells change. For example, if −3 and 10 are entered, the program prints `Result: 7`.

Common mistake

`nextInt()` reads only integers. If the user types "abc" or "7,5", the program stops with an error message: `Exception in thread "main" java.util.InputMismatchException`. The "Read num" box in the diagram assumes the user will enter data of the right type. This assumption holds in the program too; catching invalid input needs a separate decision or exception handling. For decimal numbers `nextDouble()` is used; this method expects the decimal separator set in the computer's regional settings.

1.4.5 Conditional flow

A sequential program follows the same steps every time it runs. In most problems, however, what has to be done depends on the situation. If the grade is below 50 the student fails, otherwise passes. If a number is even one message is written, if odd another. Conditional flow gives the program this flexibility.

1.4.5.1 The decision structure

In conditional flow one of two or more possibilities is chosen. The structure has three parts. The **decision point** (K) contains a condition question; the answer to the question is Yes or No. If the answer is Yes, the flow goes toward the process **P**. If the answer is No, the flow goes toward the process **Q**. When the two paths finish their work, they join at a connector and the flow continues with the next step.

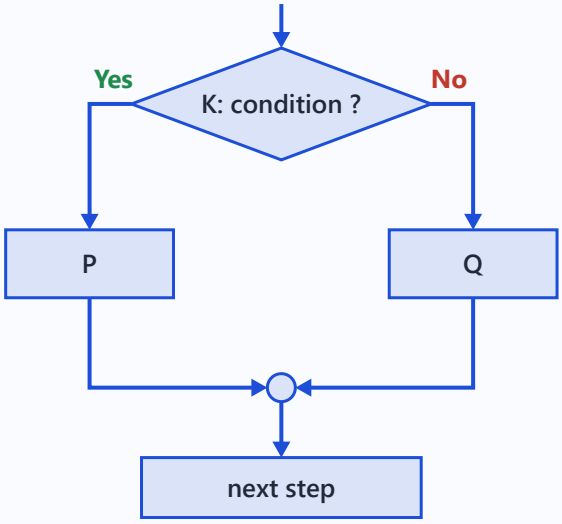


Figure 1.4.10: The general form of conditional flow. If the condition K is true, P on the Yes branch runs; if false, Q on the No branch runs. The two branches join at the connector.

The Java counterpart of this structure is the `if` / `else` statement:

```
if (K) {
    P;   // condition true
} else {
    Q;   // condition false
}
```

K in the parentheses is a `boolean` expression whose value is `true` or `false`. In Java the condition must be a `boolean`; as in C, zero or a nonzero integer cannot stand in for a condition. The curly braces gather the statements that run in a branch into a **block**. If a branch has a single statement, the braces may be omitted. But when a second statement is later added to that branch, a mistake is easily made; so always writing the braces is a good habit.

Conditional flow gives a program three things. **Flexibility:** the program behaves differently for different inputs. **Decision making:** an operation is done only when needed; for example, division by zero is attempted only if the divisor is not zero. **Control of the code:** the program checks a situation and acts accordingly; for example, it notices invalid input and writes a warning.

The decision structure also has a one-branch form. If there is no job to do on the No branch, that branch goes straight to the connector. This is a **one-way decision** and is written in Java as an `if` without `else`: `if (x > 0) System.out.println(x);`. If the condition is false, nothing is done and the flow continues with the next statement.

1.4.5.2 Is the number odd or even?

The problem is to print whether an entered integer is odd or even. A number being even means that the remainder of its division by 2 is 0. The remainder of a division is found with the **mod** operation: 7 mod 2 = 1, because 7 = 3 × 2 + 1. 8 mod 2 = 0, because 8 = 4 × 2 + 0. The algorithm has seven steps:

1. Start.
2. Read the number.
3. Take the number mod 2 and assign it to the result variable.
4. If the result is 0 go to 5, otherwise go to 6.
5. Write "even" on the screen, go to 7.
6. Write "odd" on the screen.
7. End.

The "go to ..." expressions in the algorithm stand out. In an algorithm described in words, branching can be shown only with step numbers like this. In a diagram, the same branching is drawn with a decision box and two arrows; step numbers are not needed.

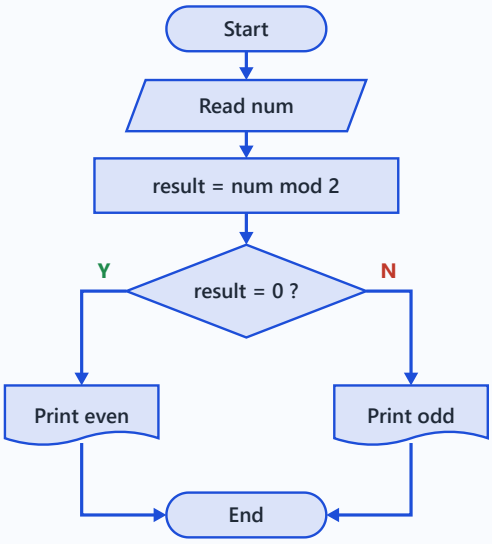


Figure 1.4.11: The odd/even diagram. The process box computes the remainder, and the decision box asks whether the remainder is 0. The Y branch writes "even", the N branch "odd"; the two branches join at End.

The Java counterpart of the diagram is as follows:

```java
import java.util.Scanner;

public class OddOrEven {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Number: ");
        int num = scanner.nextInt();
        int result = num % 2;
        if (result == 0) {
            System.out.println("even");
        } else {
            System.out.println("odd");
        }
        scanner.close();
    }
}
```

The "mod" in the diagram is the `%` operator in Java. Line 8 computes `result = num % 2`. The `if` on line 9 is the decision box and asks the question `result == 0`. If the condition is true, line 10 runs; if false, line 12; never both. The `else` on line 11 is the start of the No branch.

Table 1.4.3 summarizes two separate runs of the program with the inputs 7 and 8. When 7 is entered, `result = 7 % 2 = 1`. `1 == 0` is false, the flow goes to the No branch, that is, the `else` block, and "odd" is printed. In this run line 10 in the `if` block never runs. When 8 is entered, `result = 0`, "even" is printed from the Yes branch, and this time line 12 is skipped. In each run only one of the two branches runs; the two paths join at End.

Input	Process box	Decision	Exit	Line that runs	Output
7	7 % 2 = 1	1 == 0 → false	No	12	odd
8	8 % 2 = 0	0 == 0 → true	Yes	10	even

Table 1.4.3: Two runs of the `OddOrEven` program.

Negative numbers need separate attention. In Java the result of the `%` operation takes the sign of the dividend. So the value of `-3 % 2` is −1, not 1; and `-4 % 2` is 0. Since the diagram in the slides sets up the condition as "result = 0 ?", it is not affected by this detail: since −1 ≠ 0, "odd" is correctly printed for −3. If the condition had been set up as "result = 1 ?", the situation would change. Table 1.4.4 compares the two conditions with the inputs −3, 7 and 0.

Input	`num % 2`	Condition `== 0` (from the slides)	Condition `== 1`
−3	−1	No → odd	No → even
7	1	No → odd	Yes → odd
0	0	Yes → even	No → even

Table 1.4.4: The effect of the condition in the decision box on the result for a negative input (in Java the remainder takes the sign of the dividend).

Common mistake

Writing the odd check as `if (num % 2 == 1)` gives a wrong result for negative odd numbers. For −3, the value of `-3 % 2` is −1, so the condition is false and the program prints `-3 even`. The compiler does not catch this mistake, because the code follows Java's rules; the error is in the logic. The safe way is to test evenness: `num % 2 == 0`. Whether the remainder is −1 or 1, the number falls into the "odd" branch. When a remainder that is always non-negative is needed, as in mathematics, `Math.floorMod(num, 2)` is used; the value of `Math.floorMod(-3, 2)` is 1.

1.4.5.3 Nested decisions: positive, negative, zero

A decision box has only two exits. When there are three possibilities, two decision boxes are connected one after the other: the No exit of the first question goes to the second question. Figure 1.4.12, which finds whether a number is positive, negative or zero, shows this structure.

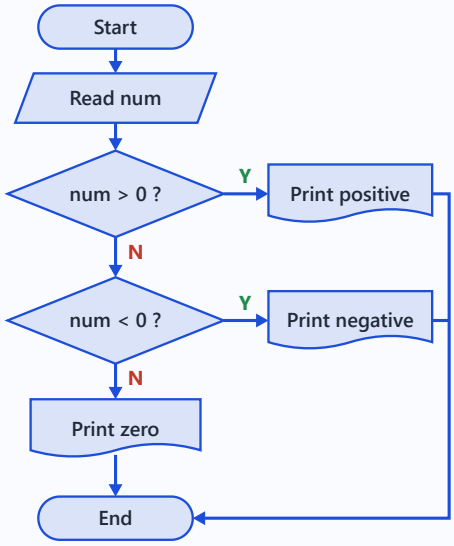


Figure 1.4.12: A nested decision. The No exit of the first decision (num > 0 ?) goes to the second decision (num < 0 ?). Of the three Print boxes, only one runs in each run.

The first decision asks the question "num > 0 ?". If the answer is Yes, "positive" is printed and the flow goes to End; the second question is never asked. If the answer is No, the number is either negative or zero. To separate these, the second decision asks the question "num < 0 ?". If Yes, "negative" is printed; if No, "zero". A number reaching the No exit of the second decision is neither positive nor negative; only one possibility is left, and that is zero. So a third question is not needed.

In Java a nested decision is written with an `else if` chain:

```
Scanner scanner = new Scanner(System.in);
int num = scanner.nextInt();
if (num > 0) {
    System.out.println("positive");
} else if (num < 0) {
    System.out.println("negative");
} else {
    System.out.println("zero");
}
```

`else if` is not a separate statement in Java. In the `else` branch of the first `if` there is a second `if` statement; the curly braces let this structure be written at a single level. That the second decision in the diagram is attached to the No exit of the first shows exactly this nesting.

Table 1.4.5 traces three runs of the program. When −4 is entered, the first condition `-4 > 0` is false; the flow goes from the No exit to the second decision. `-4 < 0` is true and "negative" is printed; the last `else` branch is skipped. When 0 is entered, `0 > 0` is false, because 0 is not positive. `0 < 0` is false too; since both conditions are false, the last `else` runs and "zero" is printed. When 5 is entered, the first condition is true and the second condition is never tested.

Input	1st decision	2nd decision	Output
−4	−4 > 0 → false	−4 < 0 → true	negative
0	0 > 0 → false	0 < 0 → false	zero
5	5 > 0 → true	not asked	positive

Table 1.4.5: Three runs of the nested decision program. The number of decisions tested is 1 or 2 depending on the input.

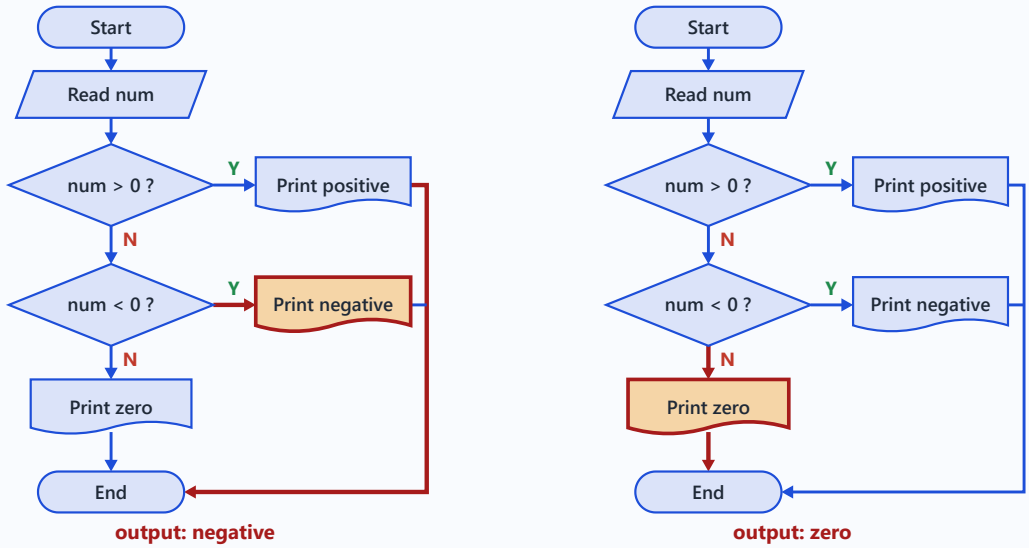


Figure 1.4.13: Two runs of the nested decision. (a) Input −4: the No exit of the first decision and the Yes exit of the second are followed and "negative" is printed. (b) Input 0: the No exits of both decisions are followed and

"zero" is printed. Highlighted arrows show the path followed.

Since there are three branches, trying the diagram completely takes at least three inputs. Each input runs a different Print box. This is the first example of the idea of *path coverage* used in choosing test cases: every exit of every decision box must be tried at least once.

Example 1.4.2

In a diagram that converts a grade between 0 and 100 into a letter grade there are four possibilities: A if grade ≥ 85, B if ≥ 70, C if ≥ 50, otherwise F. For four possibilities, three decision boxes are connected one after another. The Java counterpart is an `else if` chain:

```
if (grade >= 85) {
    System.out.println("A");
} else if (grade >= 70) {
    System.out.println("B");
} else if (grade >= 50) {
    System.out.println("C");
} else {
    System.out.println("F");
}
```

When 72 is entered, `72 >= 85` is false and `72 >= 70` is true; "B" is printed and the third decision is never tested. When 45 is entered, all three conditions are false and "F" is printed. For 72 two decisions are tested, for 45 three. In the second condition there is no need to ask separately `grade < 85`: only grades below 85 reach that decision. Boundary values also fall into the right branch: 85 → A, 84 → B, 70 → B, 50 → C, 49 → F.

1.4.5.4 Compound conditions: AND, OR

Some decisions depend on two conditions at once. For a person to be able to drive a car, they must be both 18 or older and hold a license. This is an **AND** condition. If being a student or being over 65 is enough to benefit from a discount, this is an **OR** condition.

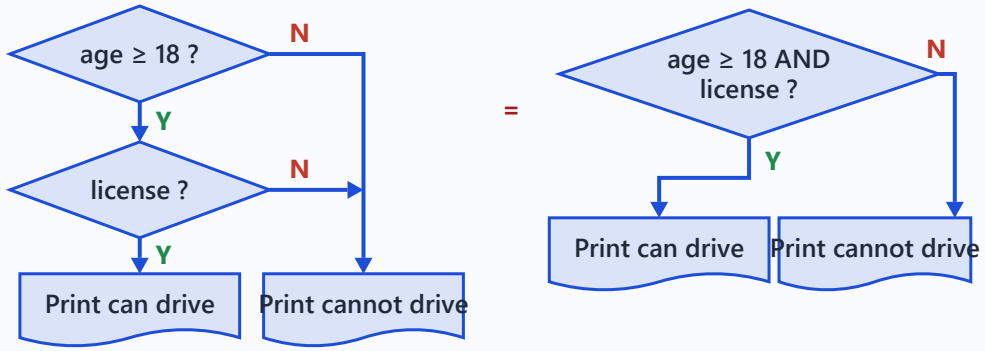


Figure 1.4.14: Two drawings of an AND condition. On the left two separate decisions are connected one after the other; the No exit of the first decision goes straight to "cannot drive". On the right the two conditions are written in a single decision box as "age ≥ 18 AND license ?". The two drawings give the same result.

Figure 1.4.14 draws the AND condition in two ways. In the drawing on the left, "age ≥ 18 ?" is asked first. If the answer is No, there is no need to check whether there is a license; the person cannot drive anyway, and the No exit of the first decision goes straight to the "Print cannot drive" box. If the answer is Yes, "license ?" is asked in the second decision. The drawing on the right fits the same logic into a single decision box. In an OR condition the situation reverses: when drawn with two decisions, the Yes exit of the first decision goes straight to the result, because it is enough for one of the conditions to be true.

In Java AND is written with `&&` and OR with `||`: `if (age >= 18 && license)`. Here `license` is a `boolean` variable. Table 1.4.6 gives the possible cases for both conditions; the table was produced with a Java program.

`age`	`license`	`age >= 18`	`age >= 18 && license`	`age >= 18 || license`
20	yes	true	true	true (license not looked at)
20	no	true	false	true (license not looked at)
16	yes	false	false (license not looked at)	true
16	no	false	false (license not looked at)	false

Table 1.4.6: Truth table of the AND and OR conditions. The note in parentheses shows the cases where Java does not look at the second condition at all.

Definition 1.4.5

Short-circuit evaluation is not evaluating the right side at all once the result is determined by the left side of the `&&` and `||` operators. In `A && B`, if A is false the result is false and B is not looked at. In `A || B`, if A is true the result is true and B is not looked at.

Short-circuiting is the Java counterpart of the diagram drawn with two decisions. In the diagram the No exit of the first decision skipped the second decision; Java too looks at the second condition only if necessary. This behavior can be seen with an experiment. If the right-hand condition is written as a method that increments a counter each time it is called, then for `age = 16` the expression `age >= 18 && hasLicense(true)` gives `false` and the counter stays at 0; the method was never called. For `age = 20` the expression `age >= 18 || hasLicense(false)` gives `true` and the counter is again 0.

Short-circuiting has a practical use: the condition on the left can guarantee that the one on the right can be computed safely. In the expression `count != 0 && total / count > 2`, if `count` is 0 the division is never done. Had the order been reversed, that is, the division put on the left, the program would stop with an `ArithmeticException` for `count = 0`.

1.4.6 Repetitive flow

One of the properties that makes a computer powerful is that it can do the same job thousands of times without tiring. Repetitive flow is the kind of flow in which operations are repeated many times. In a diagram, repetition is shown by an arrow going backward. This arrow returns to the top of a decision box; the decision box decides when the repetition ends.

1.4.6.1 Kinds of loops

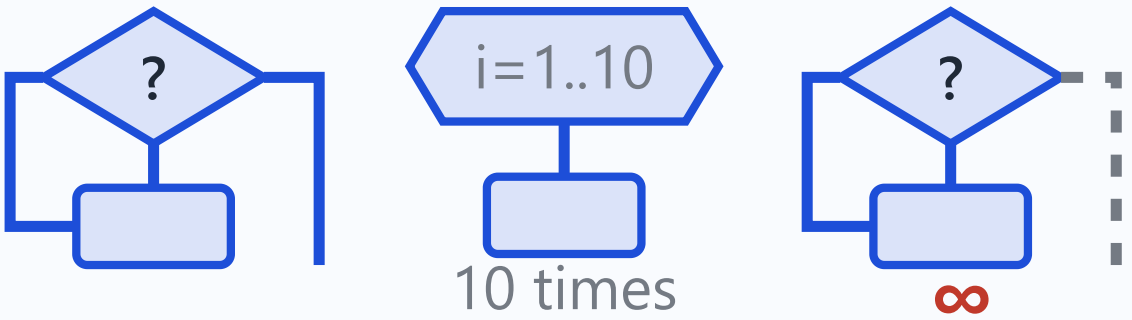


Figure 1.4.15: Icons of the three kinds of loop. (a) A condition-controlled loop. (b) Repetition a fixed number of times. (c) An infinite loop.

A **condition-controlled loop** repeats as long as a condition holds. How many times it will run may not be known in advance. "Read numbers until the user enters 0" is such a loop; how many numbers the user will enter is not known when the program is written. In Java this kind of loop is usually written with `while`.

In a loop that repeats a **fixed number of times**, the number of repetitions is known from the start. The job "print the numbers from 1 to 10" takes exactly 10 iterations. In Java this kind of loop is usually written with `for`. The choice between `for` and `while` is a matter of readability, not a necessity; every `for` loop can be written with `while` and every `while` loop with `for`.

In an **infinite loop** the condition always holds, so the operations repeat without stopping. An infinite loop is usually a bug. Forgetting to update the loop variable, setting up the condition wrongly or attaching the back arrow to the wrong place produces an infinite loop. Some programs, however, deliberately run with an infinite loop. A web server waits for incoming requests until it is shut down; the main loop of an embedded system turns as long as the device stays on. In Java this kind of loop is written `while (true)`.

Loops give a program three things. A loop is **efficient**: the same operation is done as many times as needed without writing the code over and over. A loop is **dynamic**: the number of repetitions can vary with the input and the program adapts to different situations. A loop **avoids code repetition**: to print 1000 numbers, 3–4 lines are enough, not 1000.

Every loop has three parts. The **initialization** gives the loop variable its first value and runs only once. The **condition** is tested before every iteration and decides whether the loop goes on. The **update** changes the loop variable and runs at the end of each iteration. If one of these three parts is missing or wrong, the loop either never runs, runs too few or too many times, or never ends.

1.4.6.2 Printing the numbers from 1 to 10

The first example of repetitive flow is printing the numbers from 1 to 10 on the screen. The algorithm has seven steps:

1. Start.
2. count = 1.
3. If count ≤ 10 go to 4, otherwise go to 7.
4. Write count.
5. Increase count by 1.
6. Go to 3.
7. End.

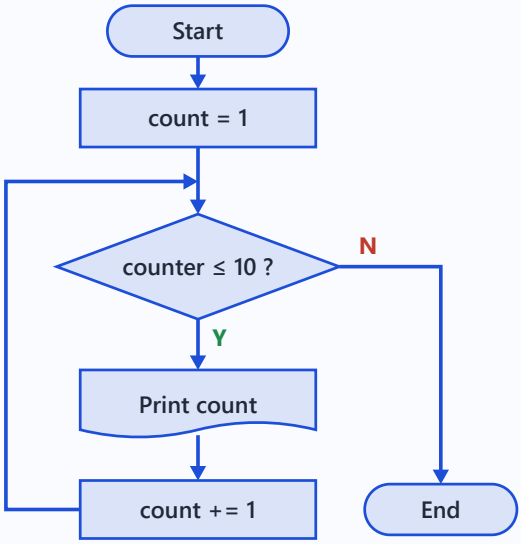


Figure 1.4.16: The diagram that prints the numbers from 1 to 10. The arrow leaving the "count += 1" box goes left, then up and returns to the top of the decision box. The No exit of the decision box is the loop's only exit door.

The "go to 3" in step 6 of the algorithm is, in the diagram, the back arrow returning to the top of the decision box. This arrow forms the loop. The decision box is the exit door of the loop: the moment the counter exceeds 10, the flow goes from the No exit to End. Note that the back arrow is attached to the decision box, below the "count = 1" box. The initial value is given only once, before the loop.

The Java counterpart of the diagram is a `while` loop:

```
public class OneToTen {
    public static void main(String[] args) {
        int count = 1;
        while (count <= 10) {
            System.out.print(count + " ");
            count += 1;
        }
    }
}
```

Line 3 is the initialization: `count` gets the value 1. The `while (count <= 10)` on line 4 is the decision box. As long as the condition is true, the body between the curly braces runs. Line 5 writes the counter; since `print` adds no line end, the numbers appear on the same line, one space apart. Line 6 is the update: `count += 1`, that is, `count = count + 1`. When the body ends, execution goes back to line 4, to the condition. This return is the back arrow in the diagram; in Java no separate "go to" statement is written, and the closing curly brace of the `while` does this job.

Table 1.4.7 is the trace of the loop. The 4th–8th iterations in the middle of the table are abbreviated because they follow the same pattern.

Iteration	`count` (start of iteration)	Condition	Printed	`count` (end of iteration)	Output
1	1	1 <= 10 → true	1	2	1
2	2	2 <= 10 → true	2	3	1 2
3	3	3 <= 10 → true	3	4	1 2 3
…	…	…	…	…	…
9	9	9 <= 10 → true	9	10	1 2 3 4 5 6 7 8 9
10	10	10 <= 10 → true	10	11	1 2 3 4 5 6 7 8 9 10
—	11	11 <= 10 → false	—	11	1 2 3 4 5 6 7 8 9 10

Table 1.4.7: Trace of the `OneToTen` program. The last row is the test in which the condition is false and the loop is left.

```
1 2 3 4 5 6 7 8 9 10
```

Two important numbers can be read from the table. The body has run 10 times; one number was printed in each iteration. The condition, however, was tested 11 times: ten times it came out true and the last time false. The condition being tested one more time than the body is a property of every loop with the decision box at the start. When the loop is left, `count` is 11, not 10. Because after 10 was printed the counter was increased once more, and then the condition came out false.

1.4.6.3 Boundary condition: < or ≤?

A single character in the loop condition changes the number of repetitions. If the counter starts at 1 and goes up by 1 each iteration, four different forms of the condition give the results in Table 1.4.8.

Condition	Body	Condition tests	`count` at exit	Last printed
`count < 10`	9	10	10	9
`count <= 10`	10	11	11	10
`count < 11`	10	11	11	10
`count <= 11`	11	12	12	11

Table 1.4.8: Number of times the body runs depending on how the condition is written, when the counter starts at 1 and goes up by 1 (measured with Java).

The condition `count < 10` does not print 10, because when the counter is 10, `10 < 10` is false. The body runs 9 times and the numbers from 1 to 9 are printed. The conditions `count <= 10` and `count < 11` mean the same; both run 10 iterations and print 10 too. `count <= 11` runs one iteration too many and prints 11 as well.

Definition 1.4.6

An **off-by-one error** is a loop running one iteration too few or one too many because its boundary condition was written wrongly.

This mistake is one of the most common in programming and the compiler does not catch it, because the wrongly written condition is also a valid Java expression. There is a simple way to prevent the mistake: test the condition *with the last value*. "From 1 to 10" includes 10. So the condition must be true when the counter is 10. `10 < 10` is false and `10 <= 10` is true; the right condition is `count <= 10`. The same check is made with the first value: the condition must be true when the counter is 1, otherwise the loop never starts.

1.4.6.4 The loop symbol and for

The same problem, printing the numbers from 1 to 10, can be drawn more briefly with the special hexagon symbol for loops. The algorithm shrinks to three steps:

1. Start.
2. Loop: from 1 to 10.
3. Write the loop variable.
4. End of loop.
5. End.

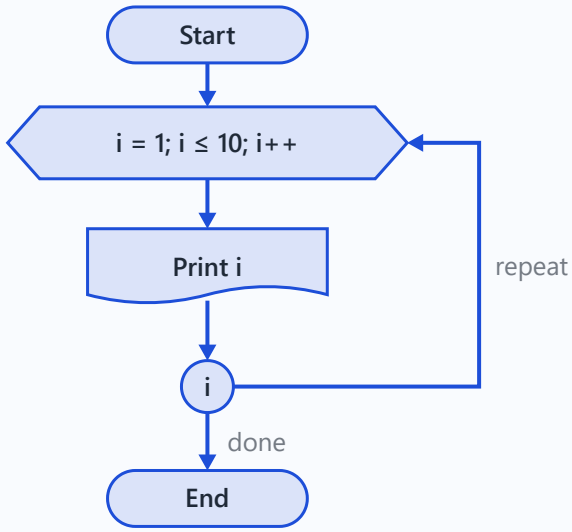


Figure 1.4.17: Printing 1 to 10 with the loop symbol. The hexagon gathers the initialization, the condition and the update in a single box. The small circle with "i" in it is the end of the loop: from there the flow returns to the hexagon along the "repeat" arrow, and when the loop is done it moves on to End along the "done" arrow.

The hexagon symbol gathers the three parts of the loop in a single box. The **initialization** is `i = 1` and runs once. The **condition** is `i ≤ 10` and is tested before every iteration. The **update** is `i++`, that is, it increases i by 1 and runs after every iteration. The small circle with "i" in it is the end of the loop. When the body ends, the flow returns from this circle to the hexagon; the hexagon does the update and tests the condition again. When the condition is false, the flow passes to End along the "done" arrow below the circle.

The Java counterpart of the hexagon is the `for` statement. The three parts are written in the same order inside the parentheses of `for`, separated by semicolons:

```
public class ForOneToTen {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            System.out.print(i + " ");
        }
        System.out.println();
    }
}
```

This program and the `while` version give the same output: `1 2 3 4 5 6 7 8 9 10`. The difference is only in the notation. Since `for` gathers the three parts of the loop in one line, the reader sees the number of repetitions at a glance. In the `while` version the initialization is written before the loop and the update at the end of the body, and they can be overlooked.

There is one more small but important difference between `for` and `while`. The variable `int i` declared in the initialization part of `for` is valid only inside the loop. If you try to use `i` after the loop, the compiler gives a `cannot find symbol` error. In the `while` version, since `count` was declared before the loop, it can be used after the loop too, and its value is 11.

1.4.6.5 A decision inside a loop: counter and accumulator

The body of a loop can contain any kind of step, a decision box included. The example problem is this: the grades of five students are read; how many students passed (grade ≥ 50) and the class average are written.

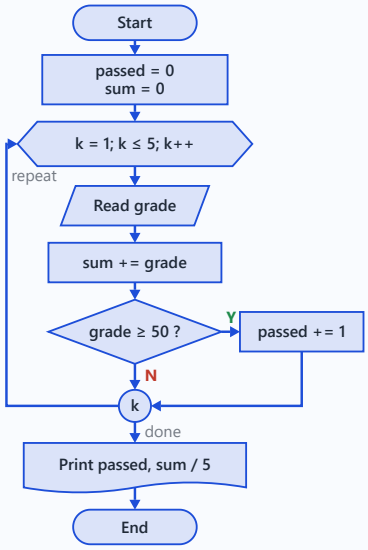


Figure 1.4.18: A decision inside a loop. The loop symbol runs for k = 1 to 5. In each iteration a grade is read and added to the sum. The Yes branch of the decision box increases the counter (passed) by 1; the No branch skips the counter and goes to the end of the loop.

```
int passed = 0, sum = 0;
for (int k = 1; k <= 5; k++) {
  int grade = scanner.nextInt();
  sum += grade;
  if (grade >= 50) {
    passed++;
  }
}
System.out.println(passed + " passed, average "
    + sum / 5.0);
```

This program uses two important loop patterns together.

Definition 1.4.7

A **counter** is a variable that increases by 1 when a condition holds and so counts how many times an event happens. An **accumulator** is a variable that gathers a result by adding a value in each iteration of the loop. Both get an initial value before the loop.

In the program `passed` is a counter: it increases by 1 with `passed++` only when `grade >= 50` is true. `sum` is an accumulator: whatever the condition, the grade read is added to it in every iteration. The initial value of both is 0; adding and counting start from zero. The decision box is inside the loop and is tested again in each iteration with the newly read grade. Table 1.4.9 is the trace of the program with the inputs 45, 70, 50, 30 and 88.

k	grade	sum	Decision	passed
1	45	45	45 >= 50 → false	0
2	70	115	70 >= 50 → true	1
3	50	165	50 >= 50 → true	2
4	30	195	30 >= 50 → false	2
5	88	283	88 >= 50 → true	3

Table 1.4.9: Trace of the counter and the accumulator. Each row shows the state at the end of one iteration.

When the loop ends, `passed = 3` and `sum = 283`. The program prints:

```
3 passed, average 56.6
```

Two Java details are hidden in this short program. The first is the grade on the boundary. A student with 50 passes, because the condition is written with `>=`. Had the condition been written `grade > 50`, the student with 50 would have failed and the result would be 2, not 3. The boundary value changes the condition by a single character; this is the form the off-by-one error takes in a decision box.

The second is the division. In the expression `sum / 5.0`, since the divisor is a decimal number, Java makes the division a decimal one: 283 / 5.0 = 56.6. Had the divisor been written `5`, two integers would be divided. In Java the quotient of two integers is again an integer; the fractional part is discarded and 283 / 5 = 56. The expression "sum / 5" in the diagram does not show this distinction; when translating into Java, attention must be paid to the types. Note also that the decimal separator appears on screen as a point (56.6); `println` always writes numbers with a point.

Besides the counter and the accumulator there is a third pattern: the **flag**. A flag is a `boolean` variable that records whether an event has happened. It is `false` at the start; when the event takes place it is made `true`, for example `found = true`. After the loop ends, the flag shows whether the event happened at least once.

1.4.6.6 A loop with a sentinel value

In the previous example the number of grades was known from the start: five. Often, however, it is not known beforehand how many values will come. A teacher sometimes enters 3 grades, sometimes 40. In this case the user announces the end of the input with a special value.

Definition 1.4.8

A **sentinel value** is a special value that announces the end of the input and is not counted as data. A value like 0, −1 or "end" is chosen.

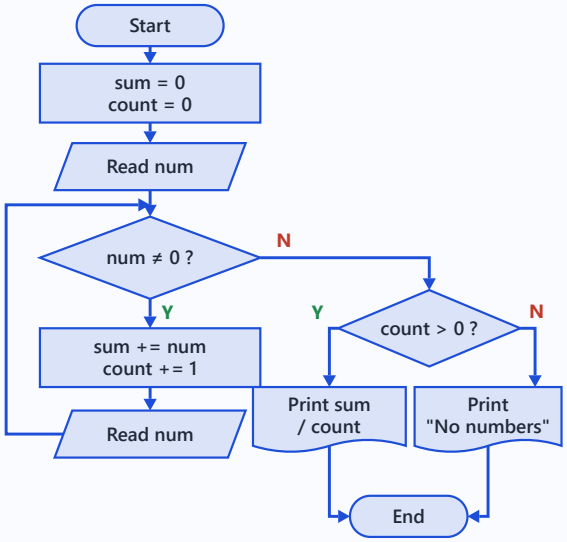


Figure 1.4.19: A loop with a sentinel value. The first number is read before the loop. At the end of the body the next number is read and the back arrow returns to the decision. The decision "count > 0 ?" after the loop separates the case where no number was entered.

```
int sum = 0, count = 0;
int num = scanner.nextInt();
while (num != 0) {
  sum += num;
  count++;
  num = scanner.nextInt();
}
if (count > 0) {
  System.out.println("Average: "
      + (double) sum / count);
} else {
  System.out.println("No numbers");
}
```

Since it is not known how many values will come, the loop is condition-controlled and is written with `while`. The most important feature of the diagram is the **read-first** arrangement. The first value is read before the loop, on line 2. The next value is read at the end of the body, on line 6. In this way every value read is tested in the condition before entering the body. When the sentinel arrives, the condition is false at once and the sentinel never takes part in the sum.

Table 1.4.10 is the trace of the program with the input 12, 7, 5, 0. In each iteration row, the `num` column shows the value added to the sum in that iteration and the new value read at the end of the body.

Step	`num`	Condition	`sum`	`count`
read first	12	—	0	0

Step	`num`	Condition	`sum`	`count`
iteration 1	12 → 7	12 != 0 → true	12	1
iteration 2	7 → 5	7 != 0 → true	19	2
iteration 3	5 → 0	5 != 0 → true	24	3
decision	0	0 != 0 → false	24	3

Table 1.4.10: Trace of the sentinel loop with the input 12, 7, 5, 0. 0 is the sentinel and does not take part in the sum.

When the loop is left, `sum = 24` and `count = 3`. The decision after the loop tests the condition `count > 0`; the condition is true and the average is printed:

```
Average: 8.0
```

Note the expression `(double) sum / count` in the average computation. `(double)` is a **cast**: it converts `sum` to a floating-point number before the division. In this way the division becomes a decimal one and 24 / 3 = 8.0. Had the cast not been written, two integers would be divided; for 24 / 3 the result would still be 8, but for 25 / 3 it would be 8 instead of 8.33….

A sentinel value must be a value that cannot appear in the real data. For student grades, −1 is a suitable sentinel; 0 is not, because a student may have scored 0 on the exam. If ordinary integers are being read, as in this example, 0 can be chosen as the sentinel, but then 0 can never be entered as data.

The `count > 0` check after the loop is not unnecessary. If the user enters 0 as the first number, the loop never runs; `sum` and `count` stay at 0. Without the check, 0 / 0 would be computed. In Java the result of this division depends on the type. `(double) 0 / 0` gives no error; it produces the value `NaN` (Not a Number) and `Average: NaN` would be printed on screen. With integers, however, `0 / 0` stops the program with `java.lang.ArithmeticException: / by zero`. Thanks to the check, the program prints `No numbers` in this case.

Common mistake

If the reading is put only at the start of the body, that is, "reading inside the loop" instead of "read first", the sentinel is added to the sum too. If the sentinel is 0, the sum does not change but `count` is counted one too many and the average comes out wrong. If the sentinel is −1, both the sum and the count are wrong. This is why the read-first arrangement exists: every value is tested before it is used.

1.4.6.7 Leaving a loop early

In a search, once the thing sought is found the remaining iterations are unnecessary. The example problem is this: five numbers are read in order; the position at which the first negative number was entered is written. Once a negative number is found there is no point in reading the remaining numbers.

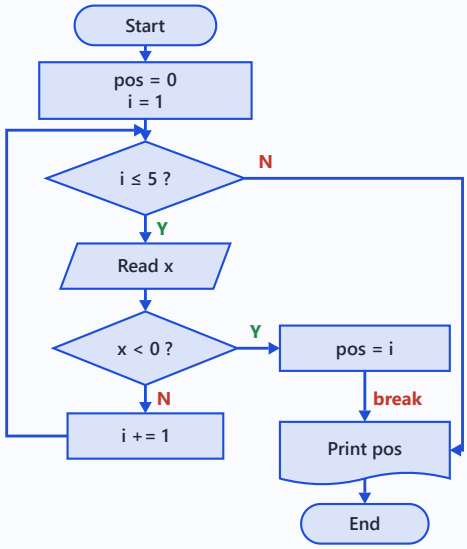


Figure 1.4.20: Leaving a loop early. The loop has two exits: the No exit of the "i ≤ 5 ?" decision (what was sought was not found) and the Yes branch of the "x < 0 ?" decision (what was sought was found, break). The two paths join at the "Print pos" box.

In the diagram there is a second decision inside the loop: "x < 0 ?". The Yes branch of this decision takes the flow straight out of the loop after the assignment `pos = i`. So the loop has two exits. The first is the usual exit: when `i` passes 5 the condition becomes false; this means no negative number was found. The second is the early exit: when a negative number is found, the loop ends at once. The two paths join at the "Print pos" box. In Java the second exit is written with the `break` statement:

```
int pos = 0;
for (int i = 1; i <= 5; i++) {
  int x = scanner.nextInt();
  if (x < 0) {
    pos = i;
    break;
  }
}
System.out.println("Position: " + pos);
```

`break` ends only the loop it is in. Execution continues with the first statement after the loop, here `println`. The initial value of the `pos` variable is 0 and means "not found". Since positions start from 1, 0 is never confused with a real position. Table 1.4.11 is the trace of the program with the input 4, 9,

−2, 6, 1.

`i`	**Loop condition**	`x`	**Inner decision**	`pos`
1	1 <= 5 → true	4	4 < 0 → false	0
2	2 <= 5 → true	9	9 < 0 → false	0
3	3 <= 5 → true	−2	−2 < 0 → true; `break`	3

Table 1.4.11: Trace of the search with `break` with the input 4, 9, −2, 6, 1. The loop ends in the third iteration; 6 and 1 are never read.

In the third iteration `x = -2` is read and `-2 < 0` is true. `pos = 3` and `break` runs. `break` also skips the update part of `for` (`i++`) and the condition; the loop ends at once. The program prints `Position: 3` . Had no negative number been entered, for example with the input 4, 9, 2, 6, 1, the loop would have run five iterations, left through the No exit of the condition when `i = 6` , and `Position: 0` would have been printed.

This diagram is not structured: the loop block has two exits. The Böhm–Jacopini theorem says the same job can also be done with a single-exit structure. The structured counterpart uses a flag instead of `break` :

```
int pos = 0;
boolean found = false;
for (int i = 1; i <= 5 && !found; i++) {
  int x = scanner.nextInt();
  if (x < 0) {
    pos = i;
    found = true;
  }
}
```

Here `found` is a flag. It is made `true` when a negative number is found. The loop condition now has two parts: `i <= 5 && !found` . The `!` operator means "not"; `!found` means "not yet found". When the flag becomes `true` , the condition becomes false and the loop ends by its usual exit. Table 1.4.12 is the trace of this version with the same input.

`i`	**Loop condition**	`x`	`found`	`pos`
1	1 <= 5 && !false → true	4	false	0
2	2 <= 5 && !false → true	9	false	0
3	3 <= 5 && !false → true	−2	true	3
4	4 <= 5 && !true → false	—	true	3

Table 1.4.12: Trace of the flag version with the same input. The loop's only exit is the condition becoming false.

The two versions give the same result: `Position: 3`. The difference between them is in the value of `i` when the loop ends. In the `break` version the loop ends while `i = 3`. In the flag version, at the end of the third iteration `i++` runs, `i = 4`, and the condition is false with this value. That is why in the flag version the position is read not from `i` but from `pos`. Which version to choose is a matter of style. In short loops `break` is readable; in long loops, having a single exit makes tracing easier.

1.4.6.8 Nested loops

A loop's body can contain another loop. This structure is called a **nested loop**. In each iteration of the outer loop, the inner loop runs through all its iterations from start to end. The following program writes a small multiplication table:

```
for (int i = 1; i <= 2; i++) {
  for (int j = 1; j <= 3; j++) {
    System.out.print(i * j + " ");
  }
  System.out.println();
}
```

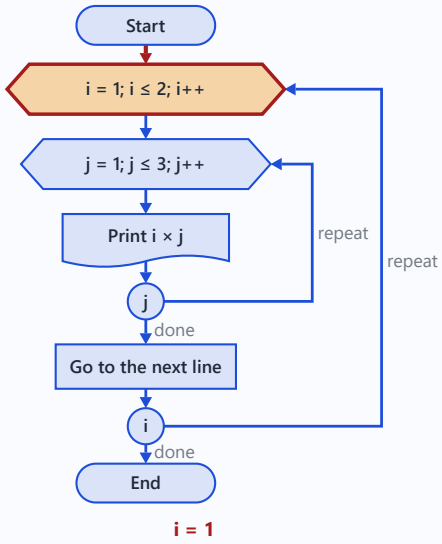


Figure 1.4.21: The diagram of a nested loop. The outer hexagon controls i, the inner hexagon j. The "j" circle of the inner loop returns to the inner hexagon, and the "i" circle of the outer loop to the outer hexagon. The "Go to the next line" box is outside the inner loop, inside the outer loop.

The outer loop runs two iterations, for `i` = 1 and 2. The inner loop runs three iterations in each outer iteration, for `j` = 1, 2, 3, and prints the value of `i * j`. When the inner loop ends, `println()` moves to the next line. Table 1.4.13 shows all the steps of the program.

`i`	`j`	Condition	Printed
1	1	1 <= 3 → true	1 × 1 = 1

`i`	`j`	Condition	Printed
1	2	2 <= 3 → true	1 × 2 = 2
1	3	3 <= 3 → true	1 × 3 = 3
1	4	4 <= 3 → false: `println()`	(next line)
2	1	1 <= 3 → true	2 × 1 = 2
2	2	2 <= 3 → true	2 × 2 = 4
2	3	3 <= 3 → true	2 × 3 = 6
2	4	4 <= 3 → false: `println()`	(next line)
3	—	3 <= 2 → false: the outer loop ends	—

Table 1.4.13: Trace of the nested loop. The inner loop starts again from `j = 1` in every outer iteration.

```
1 2 3
2 4 6
```

The point to note in the table is that in the second outer iteration `j` starts again from 1. The reason is that the initialization `int j = 1` is part of the inner `for`. Since the inner `for` statement runs from the beginning in every iteration of the outer loop, the initialization is done again each time. In the diagram this is shown by the flow entering the inner hexagon again from the top in every iteration of the outer loop.

The numbers of runs are found by multiplication. The inner body, that is, `print`, runs 2 × 3 = 6 times. `println()`, since it is part of the outer body, runs only 2 times. In general, if the outer loop runs m iterations and the inner n, the inner body runs m × n times. That is why nested loops slow down quickly on large inputs: a nested loop of 1000 × 1000 runs a million times.

1.4.7 Subprograms

As diagrams grow they no longer fit on one page and become hard to read. Some sequences of steps are also needed again and again in different places of the diagram. The solution to both problems is the same: the diagram is divided into meaningful parts.

1.4.7.1 The subprogram symbol and the method

Definition 1.4.9

A **subprogram** (predefined process) is a sequence of steps drawn as a separate diagram and called from the main diagram with the subprogram symbol. Its counterpart in Java is the **method**.

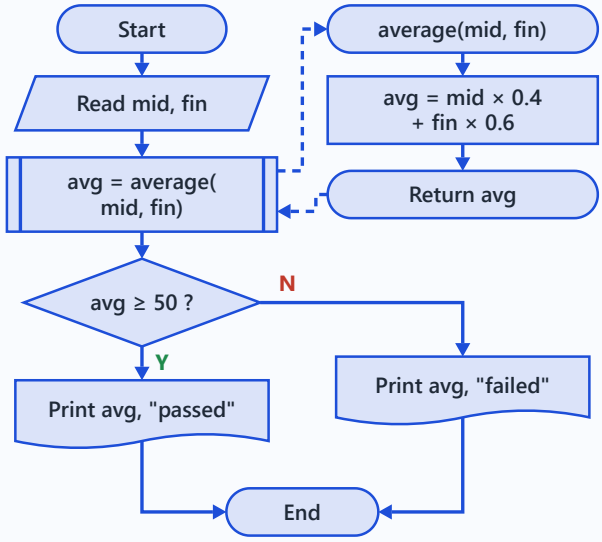


Figure 1.4.22: A subprogram. On the left the main diagram, on the right the subprogram "average(mid, fin)". The box with double side lines in the main diagram calls the subprogram. Dashed arrows show the flow passing to the subprogram and returning from the "Return avg" box to just after the call.

Figure 1.4.22 splits the diagram that computes a student's course average into two parts. The part on the right is the subprogram. In place of the Start box, the name and inputs of the subprogram are written: "average(mid, fin)". These inputs are called **parameters**. The subprogram adds 40% of the midterm grade and 60% of the final grade. At the end, instead of End there is a **Return** box: "Return avg". The Return box gives the computed value back to the place that called the subprogram.

The main diagram on the left first reads the midterm and final grades. Then it calls the subprogram with the subprogram symbol with double side lines: "avg = average(mid, fin)". At the moment of the call the flow passes to the subprogram. When the subprogram reaches the Return box, the result is given to the caller and the flow continues from just after the call. The main diagram assigns this result to the variable `avg` and, with the decision "avg ≥ 50 ?", writes whether the student passed.

A subprogram has three benefits. The same steps are not redrawn every time they are needed; they are drawn once and called many times. Each part can be tested separately; if the average computation is correct, only the reading and the decision need to be checked in the main diagram. The main diagram stays short and readable; the reader sees what the program does without going into detail.

The Java counterpart is as follows:

```
static double average(int mid, int fin) {
  double avg = mid * 0.4 + fin * 0.6;
  return avg;
}
public static void main(String[] args) {
  Scanner scanner = new Scanner(System.in);
  int mid = scanner.nextInt();
  int fin = scanner.nextInt();
  double avg = average(mid, fin);
  if (avg >= 50) System.out.println(avg + " passed");
  else System.out.println(avg + " failed");
}
```

Lines 1–4 are the subprogram. The header `static double average(int mid, int fin)` gives the name of the method (`average`), its two `int` parameters and the type of the value it returns (`double`). The statement `return avg;` is the Return box in the diagram. The expression `average(mid, fin)` on line 9 is the subprogram symbol; it is the call.

`fin` was chosen as the parameter name rather than `final`. The reason is that `final` is a reserved keyword in Java and cannot be used as a variable name. If `int final = 70;` is written, the compiler gives a series of errors, beginning with `error: not a statement`. In the diagram it is fine to write "final", but when translating into Java, keywords cannot be variable names.

1.4.7.2 Tracing a subprogram call

When the user enters 40 and 70 on the same line, the program runs in the following steps.

1. The program starts from `main`. Lines 7 and 8 read the two grades: `mid = 40`, `fin = 70`.
2. On line 9 `average` is called. The values 40 and 70 are **copied** into the parameters of the method. The method's variables `mid` and `fin` are separate from the variables of the same name in `main`; only their values are the same.
3. The subprogram's own diagram runs: `avg = 40 × 0.4 + 70 × 0.6 = 16 + 42 = 58.0`. This `avg` is the method's local variable; it is separate from the `avg` in `main`.
4. `return avg;` runs: the value 58.0 is given to the caller. The memory area reserved for the method, the method's frame, is removed from the call stack; the method's local variables disappear with it.
5. The flow continues from line 9: the returned value is assigned to `avg` in `main`.
6. The decision on line 10: `58.0 >= 50` is true; the Yes branch prints `58.0 passed`.
7. `main` ends (End).

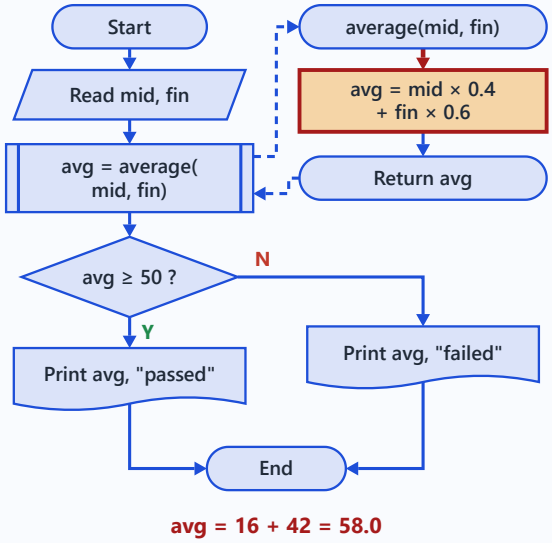


Figure 1.4.23: The moment the subprogram runs. In the main diagram the call box waits; in the subprogram the box "avg = mid × 0.4 + fin × 0.6" runs and 16 + 42 = 58.0 is computed.

The same subprogram can be called again with other inputs. Table 1.4.14 compares two runs. In the second run 45 and 30 are entered; the average comes out as 36.0 and the student fails.

`mid`	`fin`	Operation in the subprogram	Value returned	Decision	Output
40	70	40 × 0,4 + 70 × 0,6 = 16,0 + 42,0	58,0	58.0 >= 50 → true	58.0 passed
45	30	45 × 0,4 + 30 × 0,6 = 18,0 + 18,0	36,0	36.0 >= 50 → false	36.0 failed

Table 1.4.14: Two calls of the subprogram (run with Java).

Note that in the output the average is written as `58.0`, with `.0` at the end. Since the method returns a `double`, Java writes the number in decimal form; it shows the decimal part even if the value is whole. In the product `40 * 0.4`, since an `int` value is multiplied by a `double` value, the result is a `double` as well.

1.4.8 Implementation and testing

After a diagram or program is written, one has to make sure it works correctly. There are two ways: tracing the diagram or program by hand on paper, and trying it with well-chosen inputs.

1.4.8.1 The sum from 1 to n

The problem is to find the sum 1 + 2 + … + n for a number n entered by the user. In this problem all three kinds of flow come together: reading n and the initial values are sequential, the summing is repetitive, and writing the result is sequential again.

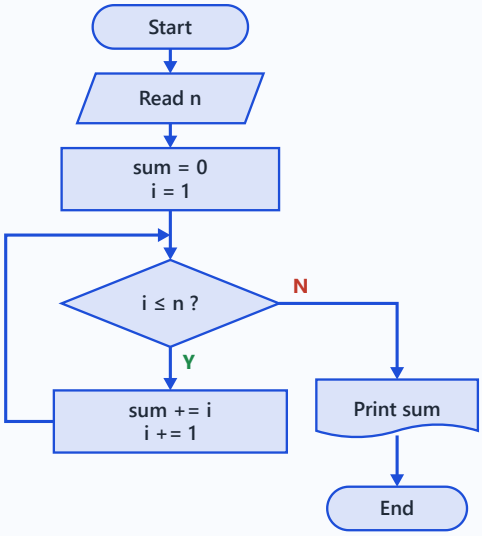


Figure 1.4.24: The sum from 1 to n. "sum = 0, i = 1" is the initialization. While the decision "i ≤ n ?" is Yes, the box "sum += i, i += 1" runs and the back arrow returns to the decision. From the No exit the sum is printed.

```
System.out.print("n: ");
int n = scanner.nextInt();
int sum = 0;
int i = 1;
while (i <= n) {
    sum += i;
    i += 1;
}
System.out.println("Sum: " + sum);
```

`sum` is an accumulator: it starts at 0 and grows by the current `i` in each iteration. `i` is a counter: it starts at 1 and increases by 1 in each iteration. Table 1.4.15 is the run in which the user enters 4.

Step	Condition and operation	`sum`	`i`
start	—	0	1
iteration 1	1 <= 4 → true; 0 + 1 = 1	1	2
iteration 2	2 <= 4 → true; 1 + 2 = 3	3	3
iteration 3	3 <= 4 → true; 3 + 3 = 6	6	4
iteration 4	4 <= 4 → true; 6 + 4 = 10	10	5
decision	5 <= 4 → false	10	5

Table 1.4.15: Trace of the sum from 1 to n for n = 4.

The body runs 4 times, the condition 5 times. The program prints `Sum: 10`; 1 + 2 + 3 + 4 = 10. For n = 5 the same diagram runs one more iteration and prints `Sum: 15`. This result can also be checked with Gauss's formula: the sum from 1 to n is n(n + 1)/2. For n = 4 one gets 4 · 5 / 2 = 10, for n = 5, 5 · 6 / 2

= 15.

The initial value of the accumulator directly affects the result. If the box "sum = 0" is mistakenly written "sum = 1", the loop still adds 1 + 2 + 3 + 4 = 10, but since the start was 1 the result comes out as 11. For addition the accumulator starts at 0, for multiplication at 1. For example, when computing n!, the product starts with `f = 1`; had it started with `f = 0`, every product would be 0. The compiler does not catch such logic errors.

1.4.8.2 Desk check and the trace table

Definition 1.4.10

A **desk check** is tracing a diagram or code on paper with an input, without using a computer. A **trace table** is the table that shows the values of the variables at each step of this trace: a column is set aside for each variable and a row for each iteration.

As an example, take a small program that adds the digits of a number. The digits of 472 are 4, 7 and 2; their sum is 13.

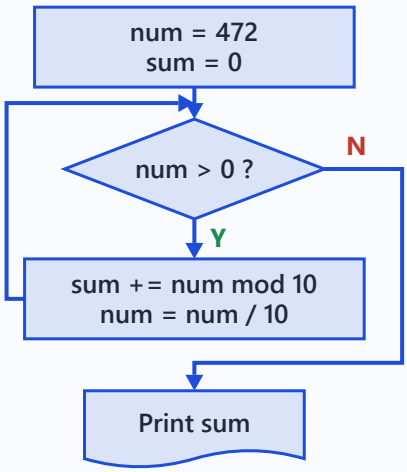


Figure 1.4.25: The digit-sum diagram. In each iteration the last digit (num mod 10) is added to the sum, and the number is divided by 10 to drop its last digit.

```
int num = 472, sum = 0;
while (num > 0) {
  sum += num % 10;
  num /= 10;
}
System.out.println(sum);
```

The program rests on two integer operations. `num % 10` gives the last digit of the number: 472 % 10 = 2. `num / 10` is integer division and drops the last digit: 472 / 10 = 47. The digits are taken in this way from right to left, one at a time. `sum += num % 10` is the short form of `sum = sum + num % 10`; `num /= 10` likewise means `num = num / 10`. Table 1.4.16 is the trace table of the desk check.

Step	`num % 10`	`sum`	`num`	Condition
start	—	0	472	472 > 0 → true
iteration 1	2	2	47	47 > 0 → true
iteration 2	7	9	4	4 > 0 → true
iteration 3	4	13	0	0 > 0 → false

Table 1.4.16: Trace table of the digit sum for 472. The condition column is the next test made with the value in that row.

The table also shows how many times the loop runs: 3 iterations and 4 tests. The program prints `13`. The real value of the table is that it shows mistakes without running the program. If the box "num = num / 10" had been forgotten, the `num` column would stay 472 in every row. The condition would always be true and `sum` would grow by 2 each iteration: 2, 4, 6, … This oddity is seen at once in the second row of the table: the loop is never going to end. Had the program been run, this mistake would have been noticed only when the screen froze.

A few habits help when building a trace table. A column is set aside for every variable; a separate column is opened for the condition too. When a variable changes, the new value is written on a new row and the old one is not erased; this way the history of the change is visible. When the loop ends, the final values of the variables are noted separately; the steps after the loop use these values.

1.4.8.3 Test cases

A diagram working correctly with a single input does not show that it will work correctly with every input. If the nested decision diagram is tried with the input 5, only the "positive" branch runs; it is not known whether the other two branches are right. That is why inputs are chosen deliberately.

Definition 1.4.11

Path coverage is running every exit of every decision box at least once in testing. **Boundary values** are the values immediately on both sides of the point where the result of the condition changes.

Table 1.4.17 shows test cases chosen for the positive/negative/zero diagram. Next to each input, the reason it was chosen and the expected output are written.

Input	Why chosen?	Expected
5	Yes path of decision 1	positive
−2	Yes path of decision 2	negative
0	No path of both decisions	zero
1 and −1	both sides of the boundary	positive, negative

Table 1.4.17: Test cases for the positive/negative/zero diagram. All expected outputs were verified with the Java program.

The first three inputs give path coverage: each of the three Print boxes runs once. The 1 and −1 in the last row are boundary values. Had the condition been mistakenly written `num >= 0` instead of `num > 0`, 5 and −2 would still have given the right result; the mistake would have shown only when 0 was entered and "positive" was printed. Off-by-one mistakes show up exactly at boundary values. For a condition such as `grade >= 50` the boundary values are 49 and 50.

For loops there is a separate rule: they are tried with inputs for which the loop runs zero times, once, and many times. Table 1.4.18 gives these cases for the sum-from-1-to-n diagram.

n	Why chosen?	Expected
0	the body never runs	Sum: 0
1	a single iteration	Sum: 1
4	the ordinary case	Sum: 10
−3	invalid input	Sum: 0 (is this what is wanted?)

Table 1.4.18: Test cases for the sum-from-1-to-n diagram (run with Java).

For n = 0, `1 <= 0` is false and the body never runs; the program prints `Sum: 0`. This is a correct result: no numbers were added. The same happens for n = −3. But here the result is debatable. A negative n is a meaningless input; instead of printing 0, it might be better for the program to give a warning. The test case brings this question to light; the answer is given by the definition of the problem.

The last rule of testing is that the expected output is found by hand *before* the trial. First it is worked out that the result for n = 4 must be 10, then the program is run and compared with the actual output. Looking at the output after running the program and saying "it looks right" is the easiest way to overlook a mistake. If there is a difference, the mistake is in the diagram, in the code, or in the expectation itself.

1.4.8.4 Ways of presenting an algorithm

A flowchart is not the only way to present an algorithm. Table 1.4.19 compares four notations on the odd/even example.

Notation	The odd/even example	Strength	Weakness
Natural language	"Read the number; if the remainder of dividing it by 2 is 0, write even, otherwise odd."	Everyone can read it	Can be ambiguous
Flowchart	Read → Process → Decision → Print even / Print odd	Shows the flow visually; branching and loops are easy to see	Clumsy for large problems
Pseudocode	`READ num; IF num mod 2 = 0 THEN PRINT "even" ELSE PRINT "odd"`	Quick to write, close to code	No standard syntax
Java	`if (num % 2 == 0) … else …`	The computer can run it	Syntax rules must be followed

Table 1.4.19: Four ways of presenting an algorithm.

Natural language is the easiest notation to read, but it is ambiguous. An expression like "if the number is large" may not say what it is larger than. Pseudocode stands between natural language and a programming language. Its keywords (READ, IF, THEN, PRINT) resemble a programming language, but it has no compiler and everyone uses their own form. Java is the most exact notation; the computer can run it. In return, all the rules, from semicolons to curly braces, must be followed. Within this range the flowchart stands out for its visuality: branches and loops are understood at once from the shape of the arrows.

In software engineering, today's counterpart of the flowchart is the UML (Unified Modeling Language) **activity diagram**. The idea is the same, the symbols are slightly different. The start is drawn with a filled circle, a decision again with a diamond, and activities with rounded boxes. In designing object-oriented programs, activity diagrams are used together with the other diagrams of UML.

1.4.8.5 Drawing tools

A flowchart can be drawn with paper and pencil or on a computer. Table 1.4.20 summarizes the common tools.

Tool	What it does	Note
Paper and pencil	The fastest sketch; the only option in an exam	Laborious to correct
draw.io (diagrams.net)	A free drawing tool that runs in the browser; ready-made flowchart symbols	Draws, does not run
Microsoft Visio	A commercial diagramming program; standard symbol libraries	Paid
Flowgorithm	Runs the diagram and tracks variables; generates code in many languages, Java included	Free, for education
Mermaid	Draws a diagram from text; GitHub and many Markdown tools display it directly	Stored together with the code

Table 1.4.20: Flowchart drawing tools.

Mermaid takes an interesting approach: the diagram is not drawn but written. The Mermaid text of the odd/even diagram is as follows:

```
flowchart TD
  A([Start]) --> B[/Read num/]
  B --> C{num mod 2 = 0 ?}
  C -- Yes --> D[Print even]
  C -- No --> E[Print odd]
  D --> F([End])
  E --> F
```

The `flowchart TD` on the first line starts a flowchart; `TD` (top-down) means a top-to-bottom layout. Each node has a letter name and a text in square brackets. The form of the brackets determines the symbol: `([ ])` the Start/End oval, `[/ /]` the input parallelogram, `{ }` the decision diamond, `[ ]` the process rectangle. `-->` is a flow line, `-- Yes -->` a labeled decision exit. The tool decides the position of the boxes on the page itself. The text form makes it easy to keep the diagram in the same files as the program code and to update it together with the code.

Whatever the drawing tool, the rules are the same: a single Start, labeled decision exits, non-crossing arrows. The tool only makes drawing easier; the correctness of the diagram still depends on the person who designs it.

1.4.9 Common mistakes

Some mistakes are very common when drawing flowcharts and when translating a diagram into Java. The compiler catches some of them; others lead to programs that compile but run wrongly. The mistakes in the second group are more dangerous, because the program produces a wrong result without any warning.

Common mistake

A single exit on a decision box. Two arrows must leave a decision symbol, and both must be labeled Y/N. If only the Yes arrow is drawn, what happens in the "no" case remains unclear. Even if there is nothing to do on the No branch, the arrow is drawn and connected straight to the junction.

Common mistake

The wrong symbol. Using a rectangle for reading from the keyboard and a parallelogram for computation makes the diagram misleading. The rule is simple: input is a parallelogram, process a rectangle, a question a diamond.

Common mistake

Using = in a comparison. `if (result = 0)` does not compile: `error: incompatible types: int cannot be converted to boolean`. Comparison is written with `==`: `if (result == 0)`.

Common mistake

Forgetting the initial value. If there is no "count = 1" box in the diagram, Java also gives an error. If `int count;` is declared and `while (count <= 10)` is written without giving it a value, the compiler gives the message `error: variable count might not have been initialized`.

Common mistake

Forgetting the update. If there is no "count += 1" box, the counter never changes and the condition stays true: an infinite loop. The compiler does not catch this mistake; the program writes 1 on the screen without stopping.

Common mistake

A boundary error. For "from 1 to 10", writing `count < 10` skips 10. The condition must be tested with the last value: when the counter is 10 the condition must be true.

Common mistake

A semicolon after the loop header. The line `while (count <= 10);` compiles, but the final semicolon is an empty statement and becomes the body of the loop. The block with curly braces below it is not part of the loop. Since the counter never increases, the condition stays true and the program hangs without printing anything. JDK 21 compiles this code without a warning.

Common mistake

Attaching the back arrow to the wrong place. In the diagram that writes 1 to 10, if the back arrow returns not to the decision box but to the top of the "count = 1" box, the counter becomes 1 again in every iteration. The condition stays true and 1 is written on the screen

without stopping: again an infinite loop. The back arrow must return to the decision box, after the initial value. In Java this mistake corresponds to writing the initial assignment inside the loop.

Common mistake

Integer division. If the expression "sum / 5" in the diagram is written in Java between two `int` s, the decimal part is discarded: 283 / 5 = 56. For a decimal result, one of the operands must be a `double` : `sum / 5.0` or `(double) sum / count` .

1.4.10 Summary

- An **algorithm** is the ordered, finite steps that solve a problem. A **flowchart** is the drawing of an algorithm with symbols and arrows; it is a plan independent of the programming language. It becomes clumsy in large programs; then pseudocode is more practical.
- Basic symbols: oval Start/End, parallelogram input, rectangle process, diamond decision, hexagon loop, wavy box print, circle connector, arrow flow line. Additional symbols: subprogram, off-page connector, data store, manual input, comment, preparation.
- Drawing rules: a single Start, at least one End; flow from top to bottom and left to right; only the decision box has two exits and the exits are labeled Y/N; lines do not cross.
- Java counterparts: Start/End → `main` ; input → `nextInt()` ; process → assignment; decision → `if` / `else` ; decision + back arrow → `while` ; hexagon → `for` ; print → `println` ; subprogram → method. The "=" in a process box in the diagram is assignment (`=`), in a decision box it is comparison (`==`).
- **The Böhm–Jacopini theorem:** every algorithm can be written with sequence, selection and repetition. In a **structured diagram** every structure has a single entry and a single exit. `goto` cannot be used in Java; `break` and `continue` are limited jumps that affect only the loop they are in.
- In **sequential flow** every step runs exactly once; the order of dependent steps cannot be changed. Java catches the reading of a variable that has received no value at compile time.
- In **conditional flow** P or Q runs according to the decision box K. Three or more possibilities are drawn with nested decisions (`else if`). In Java the result of the `%` operation takes the sign of the dividend: `-3 % 2 == -1` . AND is written `&&` and OR `||` , and they are evaluated with short-circuiting.
- In **repetitive flow** the back arrow returns to the decision box. The three parts of a loop are the initialization, the condition and the update. If the decision comes first, the condition is tested one more time than the body runs (1 to 10: body 10 times, condition 11 times).

- Loop patterns: a **counter** increases by 1 when a condition holds; an **accumulator** adds up a value in each iteration (starts at 0 for addition, 1 for multiplication); a **flag** records whether an event has happened. In a **sentinel loop** reading is done first; the sentinel does not take part in the sum; after the loop the case count = 0 is checked separately.
- **Early exit** is written with `break`; its structured counterpart is widening the loop condition with a flag. In a **nested loop** the inner body runs m × n times.
- A **subprogram** is a sequence of steps drawn and called separately; it takes parameters and returns a value with the Return box. In Java it is a method; parameters are copied, and local variables disappear when the call ends.
- In a **desk check** a diagram or code is run on paper with a trace table. **Test cases** include every exit of every decision (path coverage) and the boundary values. Loops are tried with inputs that make them run zero times, once and many times; the expected output is found by hand before the trial.
- Pitfalls: the < / ≤ boundary error; forgetting the update or attaching the back arrow to the initialization box (an infinite loop); a semicolon after the loop header; integer division.

1.4.11 Key terms

The key terms of this chapter are listed below.

algorithm flowchart pseudocode sequential flow conditional flow structured flowchart

nested decision comparison operator logical operator short-circuit evaluation loop while loop

for loop infinite loop accumulator sentinel value break / continue nested loop

subprogram (predefined process) desk check trace table path coverage

---

1.4.12 Test Yourself

1. Which symbols are used to draw the steps "save the results to a file", "write the sum on the screen" and "i = 1; i ≤ 5; i++"?

   Answer

   Saving to a file is writing to a lasting place: the data store (cylinder). Writing to the screen is drawn with the Print symbol (the box with a wavy bottom edge); some sources use the output parallelogram too. The expression that contains the initialization, condition and update together is the loop symbol (hexagon); its Java counterpart is `for (int i = 1; i <= 5; i++)`.

2. After the fragment `int x = 50, steps = 0; while (x > 1) { x = x / 2; steps++; }` has run,

what are `x` and `steps`?

Answer

With integer division x takes these values: 50 → 25 → 12 → 6 → 3 → 1. At each division `steps` increases by 1; 5 divisions are made. When x = 1, `1 > 1` is false and the loop ends. Result: `x = 1`, `steps = 5`. In the divisions 25 / 2 and 3 / 2 the decimal part is discarded.

3. How many times does the loop `for (int i = 3; i <= 15; i += 4) sum += i;` run? If `sum` starts at 0, what is it at the end?

Answer

`i` takes the values 3, 7, 11 and 15; 15 also satisfies the condition because the condition is written with `<=`. The next value is 19 and `19 <= 15` is false. The loop runs 4 times; `sum = 3 + 7 + 11 + 15 = 36`. The condition is tested 5 times.

4. The user enters 8, 0, 5, 0 in order into the average program whose sentinel value is 0. What does the program print? What happens to the value 5?

Answer

The first value read is 8; `8 != 0` is true, `sum = 8`, `count = 1`. At the end of the body 0 is read; 0 is the sentinel and the loop ends. Since `count > 0` is true, `Average: 8.0` is printed. 5 and the second 0 are never read; the program ends without waiting for them. The sentinel determines exactly where the input ends.

5. The program that finds the position of the first negative number runs with the input 3, −1, −4, 2, 7. What do the `break` version and the flag version print? What is `i` in the two versions when the loop ends?

Answer

The second number −1 is negative; both versions print `Position: 2`. −4 is not considered at all, because what is sought is the *first* negative number. In the `break` version the loop ends while `i = 2`. In the flag version, `i++` runs at the end of the second iteration; `i = 3` and `3 <= 5 && !true` comes out false. (For the comparison in the question, `i` is assumed to be declared before the loop.)

6. The outer loop runs `a` from 1 to 3. The inner loop runs `b` from 1 to `a` and its body writes one `*`; after the inner loop `println()` runs. What is seen on the screen? How many stars are

written?

Answer

The number of iterations of the inner loop depends on the outer variable: 1 for a = 1, 2 for a = 2, 3 for a = 3. Three lines are seen on the screen: `*`, `**`, `***`. In total 1 + 2 + 3 = 6 stars are written. Here the number of runs of the inner body is not m × n but the sum of the iteration counts.

7. For the job "write the multiples of 5 from 1 to 100", `for (int i = 5; i < 100; i += 5)` has been written. How many numbers are written? What is the mistake?

Answer

5, 10, ..., 95 are written: 19 numbers. When i = 100, `100 < 100` is false and 100 is not written. This is an off-by-one mistake. If the condition is tested with the last value, the mistake shows: 100 is also a multiple, so the condition must be `i <= 100`. The corrected loop writes 20 numbers.

8. In the odd/even program the condition has been written `num % 2 == 1`. What is written when −7 is entered? What would be written if the condition were `num % 2 == 0`?

Answer

In Java the remainder takes the sign of the dividend: `-7 % 2 == -1`. In the first condition `-1 == 1` is false and −7 is taken to be "even"; this is wrong. In the second condition `-1 == 0` is false, "odd" is written from the No branch; this is right. Testing evenness is safe with negative numbers too.

9. Which inputs should be chosen at the least to test the diagram "if grade ≥ 50 write 'passed', otherwise 'failed'"? Why?

Answer

For path coverage both branches must run: two grades, one ≥ 50 and one < 50. For boundary values, the two sides of the point where the result of the condition changes are chosen: 49 (failed) and 50 (passed). These two inputs also cover both branches. The ends of the valid range (0 and 100) can be added too. If the condition was mistakenly written `> 50`, only the input 50 exposes the mistake.

1.4.13 Further reading

- P. Deitel and H. Deitel, *Java: How to Program* (11th ed., 2017), Chapters 4 and 5: presents Java's control statements together with UML activity diagrams; works through counters, sentinel values and nested control structures with many examples.
- A. B. Downey and C. Mayfield, *Think Java* (2nd ed., 2019): offers a short, plain introduction to conditionals, loops and methods; the whole book can be read free on the authors' site.
- C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules" (1966): the original paper on the structured programming theorem; it proves that flowcharts can be reduced to three structures.
- Oracle, *The Java Tutorials*, "Control Flow Statements": the official explanation and short examples of the `if`, `switch`, `while`, `do-while`, `for`, `break` and `continue` statements.

1.4.14 References

1. ISO 5807:1985, *Information processing — Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts*.
2. C. Böhm, G. Jacopini, “Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules”, *Communications of the ACM* 9(5), 1966.
3. P. Deitel, H. Deitel, *Java: How to Program*, 11th ed., Pearson, 2017.
4. A. B. Downey, C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019.
5. Oracle, *The Java Tutorials*: “Control Flow Statements” (docs.oracle.com/javase/tutorial).
6. K. Sierra, B. Bates, T. Gee, *Head First Java*, 3rd ed., O'Reilly, 2022.

PART I · CHAPTER 2

Programming and Java Syntax

A programming language is made of statements that follow precise rules of syntax and meaning. The chapter first gives a general picture of how programming languages evolved, the difference between syntax and semantics, and the roles of expressions, variables and types. On the Java side it covers compiling and running source code, primitive data types, identifiers and keywords, assignment, constants, type conversions and operators in detail. A short introduction to control flow closes the chapter.

NOTES IN THIS CHAPTER

2.1 Programming

2.2 Syntax Rules

PART I · CHAPTER 2.1

Programming

However precisely an algorithm is written on paper, a computer can run it only once it is expressed in a programming language. A programming language is the bridge between the solution a person has in mind and the operations a machine performs, and these notes describe how that bridge is built. The first sections classify languages by level of abstraction, application area, execution model, and paradigm, and explain how Java compiles source code to bytecode and runs it on a virtual machine. Next come the three layers of every language: syntax, static semantics, and semantics. These layers also separate compile-time errors, run-time errors, and logic errors. The following sections show how expressions are evaluated by operator precedence, how integer division works, what static and dynamic typing mean, how operations across types behave, and how string concatenation proceeds. The last section turns an algorithm into a Java program, distinguishes expressions from statements, and shows how statement order determines the result.

LEARNING OUTCOMES

Programming languages
Define a programming language and classify languages by level of abstraction, application area, execution model, and paradigm.

How Java runs
Explain how source code is compiled to bytecode by `javac` and executed by the JVM.

Syntax and semantics
Distinguish syntax, static semantics, and semantics; recognize compile-time, run-time, and logic errors.

Expressions
Find the value of an expression using operator precedence, left-to-right evaluation, and integer division.

Variables and types
Declare variables and trace an assignment in two steps; explain static and dynamic typing and operations across types.

From algorithm to program
Tell an expression from a statement; translate an algorithm into a Java program and show why statement order matters.

2.1.1 Programming languages

People talk to each other in natural languages. English, Turkish, and German are rich languages: the same thought can be put many ways, and the listener fills in whatever is left out from context. A computer does no such filling in. Every instruction it receives must be read in exactly one way and mean exactly one thing. Programming languages were born from that need. This section defines a programming language and classifies languages by four criteria: level of abstraction, application area, execution model, and paradigm. Paradigms get a section of their own.

2.1.1.1 What is a programming language?

Definition 2.1.1

A **programming language** is an artificial language designed to express computations that a machine can carry out. Text written in such a language is a **program**, and a program in the form in which it is written is called **source code**.

The word “artificial” matters. Natural languages develop on their own over centuries, and their rules are written down afterwards by observation. Programming languages are designed up front. A document fixes precisely which words may be used, in what order they may be arranged, and what each arrangement means. For Java that document is *The Java Language Specification*.

A programming language serves two purposes. The first is to write programs that control the behavior of a machine, which requires algorithms to be expressed clearly and exactly. The second is easier to overlook: a program is a means of communication between programmers. Someone else will read the code years later, change it, or hunt for a bug in it. A good program does not merely run correctly; it is also easy to read.

Computers, phones, and embedded devices get their functions from programs. The washing steps of a washing machine, the decisions of a car’s brake control system, and the response to every touch on a phone screen are all instructions written in a programming language. In that sense a programming language is the written form of algorithms and logical processes.

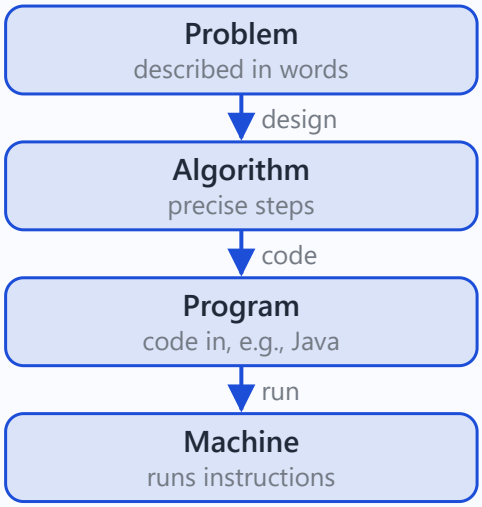


Figure 2.1.1: From problem to machine. A description in everyday language becomes an algorithm, the algorithm becomes a program, and the program runs on the machine.

Figure 2.1.1 shows the four stops along this path. Each step has its own difficulty. Going from problem to algorithm removes ambiguity and breaks the solution into precise steps. Going from algorithm to program writes those steps according to the rules of a language. Going from program to machine is handled by helper programs called compilers or interpreters. Most of these notes concern the second step: how to write an algorithm correctly in Java, and what the code you have written actually means.

There are many languages: C, C++, Java, Python, Prolog, Haskell, Scala, and hundreds more. None is best for every job, so the choice depends on the needs of the project. Speed, reliability, development time, the hardware to be used, and the experience of the team all play a part. The next subsections give classifications that help with this choice.

2.1.1.2 Levels of abstraction

A processor understands only its own instruction set. These instructions are encoded as binary numbers, and each does something very small: it moves a value from one register to another, adds two registers, or reads a word from memory. The form the processor executes directly is called **machine language** or **machine code**. Machine language is specific to a processor family; the machine code of an Intel processor does not run on an ARM processor.

Definition 2.1.2

A **low-level language** is close to the hardware: each line written in it corresponds to one or a few processor instructions. Machine language and **assembly language** belong to this group. A **high-level language** hides the details of the hardware, and the programmer thinks in terms of variables, expressions, loops, and methods. Java, Python, and C# are high-level languages.

Assembly language gives readable abbreviations to machine instructions. Names such as `ADD`, `MOV`, and `LOAD` replace the binary codes. Even so, each line still corresponds to a single machine instruction. A simple operation such as `a = b + c` takes roughly three lines of assembly: load `b` into a register, add `c`, and store the result at the memory address of `a`. In Java the same job is a single line, and the programmer does not care which register is used.

As the level of abstraction rises, programs become shorter and more readable, and they are easier to move to other hardware. In exchange, the programmer gives up some control over the hardware. In electrical and electronics engineering both ends have their place. A microcontroller's interrupt routine, or a driver with a very tight timing constraint, is sometimes written in assembly or C. An application that processes measurement data, has a user interface, or communicates over a network is usually written in Java or Python.

Level	Example	What does one line correspond to?	Portability
Machine language	binary instruction codes	a single processor instruction	only that processor family
Assembly	`ADD`, `MOV`	a single processor instruction (with a readable name)	only that processor family
High level	Java, Python, C#	many processor instructions	anywhere there is a compiler or virtual machine

Table 2.1.1: Comparison of levels of abstraction.

C sits somewhere in the middle of this classification. It is a high-level language, but it allows memory to be accessed by address and low-level code to be written. That is why it is so common in embedded systems. Java, by contrast, keeps the programmer entirely away from memory addresses and leaves memory management to a component called the garbage collector.

2.1.1.3 General-purpose and domain-specific languages

The second way to classify languages is by application area.

Definition 2.1.3

A **general-purpose language** offers a broad core set of commands that supports a wide variety of applications. A **domain-specific language** focuses on a particular kind of application and makes the operations of that kind easy.

Java, C, C++, and Python are general-purpose languages. The same language can be used to write a desktop program, a web server, a mobile app, or an embedded system. Two familiar domain-specific languages are MATLAB and SQL. MATLAB is designed for numerical computing: matrix and vector operations are built into the language and optimized to run fast. SQL (Structured Query Language) is for requesting data from a database and updating it.

The same task shows the difference best. Multiplying two $n \times n$ matrices is a single line in MATLAB:

```
C = A * B;   % one line
```

In Java the same job takes three nested loops written by hand:

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
```

In the Java code the innermost line builds the sum $C_{ij} = \Sigma_k A_{ik} \cdot B_{kj}$ step by step. The two outer loops walk over every cell of the result matrix, and the innermost loop adds up the products for that cell. The notation `C[i][j] += …` means "add to `C[i][j]`"; it assumes the array `C` starts out filled with zeros, which holds in Java because the elements of a newly created `double` array are already 0.0.

The costs of the two versions differ as well. Three nested loops of n iterations each perform n^3 multiplications and additions in total. The `A * B` in MATLAB ends up doing a similar number of operations, but it hands them to a ready-made library that uses the cache well and often runs in parallel. A domain-specific language is therefore both shorter and, in its own area, often faster. The price is flexibility: writing a web server in MATLAB is much harder than writing one in Java.

Common mistake

Writing `C = A * B;` in Java out of MATLAB habit. In Java the `*` operator is defined only for numbers; it does not know how to multiply two arrays. The compiler rejects the line with the message `bad operand types for binary operator '*'`. In Java, matrix multiplication is done either with loops or with a library method.

2.1.1.4 Compiled and interpreted languages

The third classification looks at how source code is executed. A processor does not understand source code directly; the code must first be translated into machine language. There are two basic ways to do this.

Definition 2.1.4

A **compiler** is a program that translates the whole source code into another language, usually machine code, before the program is run. An **interpreter** reads the source code and executes its commands one after another, on the spot; it does not produce a complete translation in advance. By these two routes languages are called **compiled** or **interpreted**.

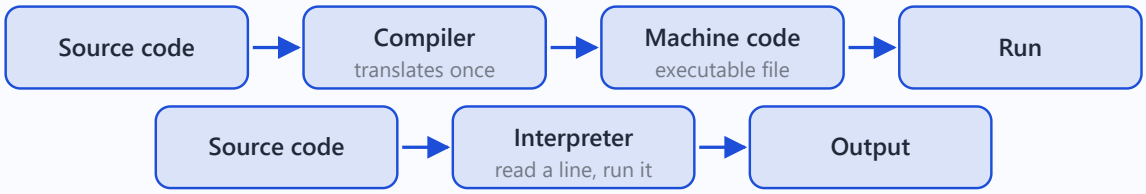


Figure 2.1.2: Execution models. Top: a compiled language (C, C++, Go, Rust): the compiler translates the source code to machine code in one pass, and the resulting executable file is run afterwards. Bottom: an interpreted language (Python, JavaScript): the interpreter reads the source code line by line and executes it directly.

Figure 2.1.2 puts the two routes side by side. In a compiled language the translation is done once and the result is written to a file; a Windows `.exe` file is such an output. The program is not translated again each time it runs, and the machine code goes straight to the processor. Because the compiler sees the whole program at once, it can optimize the code: it discards unnecessary operations and keeps frequently used values in registers. Compiled programs are therefore generally fast.

An interpreted language has no separate translated file. The interpreter keeps running alongside the program, decoding and carrying out each command when its turn comes. This approach gives flexibility: code can be tried as soon as it is written, and new commands can even be entered while the program runs. The price is speed. If the same loop runs a thousand times, the interpreter decodes the same line a thousand times.

	Compiled	**Interpreted**
How?	All of the source code is translated into object code or machine code by the compiler before the program runs.	The interpreter reads the source code and executes the commands one after another on the spot.
Speed	Fast, optimized execution	Generally slower
Errors	Many are found during compilation	Most show up only when the line is executed
Distribution	A separate build is needed for each operating system and processor	The same source code runs anywhere there is an interpreter

Table 2.1.2: Comparison of compiled and interpreted languages.

The third row of the table may be the most important difference for a programmer. The compiler reads the whole program before it runs. A misspelled word, a missing parenthesis, or an incompatible operation is caught in that reading, and the program never runs at all. In an interpreted language a faulty line goes unnoticed until the flow of the program reaches it. A Python program can run without trouble for an hour and then stop because of a typo in an `if` branch that is rarely taken.

2.1.1.5 Java's path: bytecode and the JVM

Java combines the two approaches. The source code is compiled first, but not directly to the machine code of a particular processor: it is compiled to an intermediate code called **bytecode**. The **Java Virtual Machine** (JVM) then executes this intermediate code.

Definition 2.1.5

Bytecode consists of platform-independent instructions produced by the Java compiler `javac` and stored in `.class` files. The **JVM** is the virtual machine that interprets bytecode and translates its frequently executed parts to machine code at run time with a **JIT compiler** (just-in-time compiler).

A Java program is run with two commands: the first compiles, the second executes.

```
> javac Paradigm.java      (creates Paradigm.class)
> java Paradigm
Imperative: 15
Functional: 15
```

The command `javac Paradigm.java` reads the source file. If there are no errors, it creates `Paradigm.class` in the same folder. The command `java Paradigm` starts the JVM and has it load that class. Note that the second command has no file extension: the JVM is given a class name, not a file name.

You can see what bytecode looks like with the `javap -c` tool that ships with the JDK. Here is a small method that adds two integers:

```
static int add(int a, int b) {
    return a + b;
}
```

`javac` translates this method into the following four bytecode instructions:

```
static int add(int, int);
  Code:
     0: iload_0
     1: iload_1
     2: iadd
     3: ireturn
```

The instructions operate on a stack. `iload_0` pushes the first parameter (`a`) and `iload_1` the second (`b`). `iadd` takes the two integers on top of the stack, adds them, and pushes the result back. `ireturn` returns the value on top of the stack as the method's result. The leading `i` in each instruction shows that it works with `int` . These instructions belong to no real processor; they are an abstract instruction set defined for the JVM.

This intermediate layer has a great benefit. The same `.class` file runs unchanged on Windows, Linux, or macOS, on an Intel or an ARM processor. A separate JVM has been written for each platform, while the program is compiled only once. The slogan used when Java was introduced, “write once, run anywhere,” describes this property.

The JIT compiler solves the speed problem. The JVM first runs the program by interpreting it, and counts which methods are called often. A method that runs frequently is compiled at run time into the real machine code of that machine, and later calls run directly on the processor without interpretation. As a result, long-running Java programs reach speeds close to those of compiled languages. For short programs, the JVM’s startup time is noticeable.

Java’s compilation step also keeps the most important advantage of compiled languages. `javac` reads the whole program, checks the types, and never produces a `.class` file for a faulty program. A misspelled variable name or a string assigned to a number is caught before the program runs. Other languages, such as Kotlin and Scala, also compile to JVM bytecode; they share the same virtual machine and Java’s libraries.

Common mistake

Trying to run the program as `java Paradigm.class`, or with a file whose name differs from the class name. The `java` command expects a class name; if you write the `.class` extension, the JVM looks for a class called `Paradigm.class` and cannot find it. Also, the source file of a `public` class must have the same name as the class: `public class Paradigm` compiles only in a file named `Paradigm.java`.

2.1.2 Programming paradigms

Two programmers can solve the same problem in very different ways, even in the same language, and languages themselves are designed to make one way of solving problems easier than another. The way of thinking a language favors is described by its paradigm. This section introduces four basic paradigms and illustrates each with a short program.

2.1.2.1 The idea of a paradigm

Definition 2.1.6

A **programming paradigm** is a fundamental approach to solving a problem. A paradigm determines what a program consists of (commands, functions, rules, or objects) and how the computation proceeds.

Four paradigms are mentioned often in this course. Table 2.1.3 summarizes them.

Paradigm	Core idea	Example languages
Imperative	The programmer tells the machine, step by step, *how* to change its state.	FORTRAN, BASIC, Pascal, C
Functional	Computation is the evaluation of mathematical functions; changing state is avoided.	Lisp, Scheme, Haskell
Logic	The logic of the computation is described with facts and rules; the flow of control is not spelled out.	Prolog
Object-oriented	Related data and methods are gathered under an object; commands are grouped together with the state they work on.	C++, Java, C#, Python

Table 2.1.3: Four basic programming paradigms.

The word “state” in the table is the key to telling the paradigms apart. State is the values of a program’s variables at a given moment. In an imperative program these values change at every step, and the result is the sum of those changes. In a functional program values are computed once and never changed; the result comes from composing functions. In a logic program the programmer writes neither steps nor functions, only what is true. In an object-oriented program state does not disappear; it is placed inside objects.

Languages that support more than one paradigm are called **multi-paradigm** languages. Java is fundamentally object-oriented and imperative. Since Java 8 it also includes functional elements such as lambda expressions and streams. Python and C# are multi-paradigm as well. A paradigm is therefore often a property not of a language but of a program, or of a part of a program.

2.1.2.2 Imperative and functional: the same problem

The problem is simple: find the sum of the numbers from 1 to 5. The Java program below computes it in two ways.

```
import java.util.stream.IntStream;

public class Paradigm {
    public static void main(String[] args) {
        // Imperative: change state step by step
        int sum = 0;
        for (int i = 1; i <= 5; i++) {
            sum = sum + i;
        }
        System.out.println("Imperative: " + sum);

        // Functional (Java 8+): say what you want
        int result = IntStream.rangeClosed(1, 5).sum();
        System.out.println("Functional: " + result);
    }
}
```

When the program runs, both ways give the same result:

```
Imperative: 15
Functional: 15
```

The imperative part spells out *how* the sum is found. A variable named `sum` starts at 0. The `for` loop counts `i` from 1 to 5 and on each pass updates `sum` with the statement `sum = sum + i`. The result is whatever value remains in the variable after five updates. Here the programmer thinks in the machine's place: which variable changes, in what order, and how?

The functional part states *what* is wanted. `IntStream.rangeClosed(1, 5)` produces a stream of the numbers 1, 2, 3, 4, and 5; the word `closed` says the upper bound is included. `.sum()` adds up all the numbers in that stream. No variable is updated step by step in the code; `result` is assigned only once, with its final value. The order in which the additions happen is left to the library.

The first line of the program, `import java.util.stream.IntStream;`, tells the compiler where the `IntStream` class lives. Without it the compiler would not recognize the name `IntStream` and would report `cannot find symbol`. Classes such as `System` and `String` are in the `java.lang` package, so they need no import.

2.1.2.3 Code trace: the imperative sum

The best way to see how state changes in the imperative approach is to trace the code line by line. Table 2.1.4 gives, for each pass of the loop, the value of the condition and the new value of `sum`.

`i`	`i <= 5`	**Body**	`sum`
—	—	start	0
1	true	0 + 1	1
2	true	1 + 2	3
3	true	3 + 3	6
4	true	6 + 4	10
5	true	10 + 5	15
6	false	loop ends	15

Table 2.1.4: Trace of the imperative sum. Each row shows a condition check followed by the body's execution.

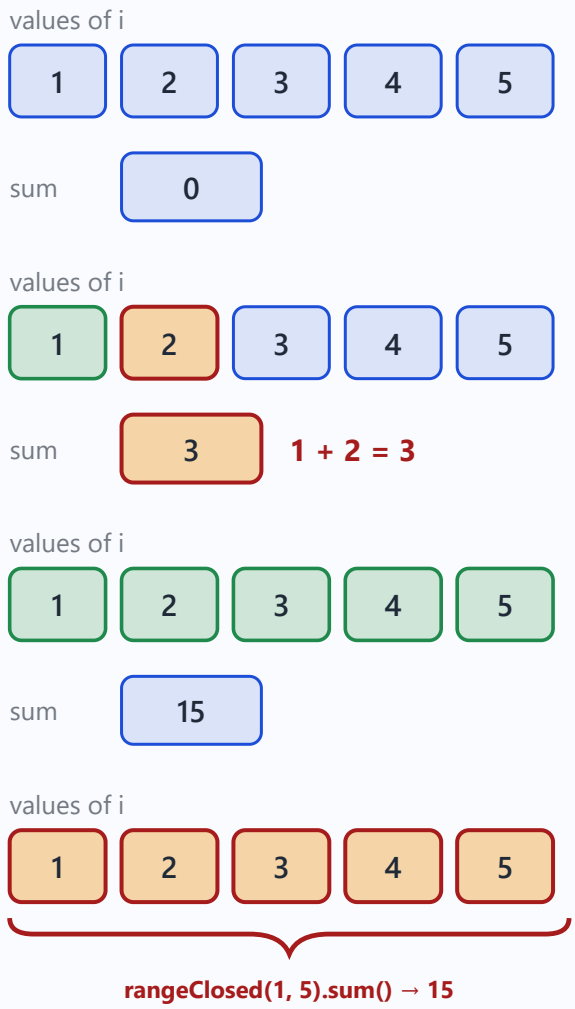


Figure 2.1.3: Four moments from the trace of the sum program. (a) The state variable `sum` starts at 0. (b) `i` = 2: 1 + 2 = 3; the first pass is marked in green. (c) When `i` becomes 6, 6 <= 5 is false and the loop ends; the sum accumulated over five passes is 15. (d) The functional route: `rangeClosed(1, 5)` produces the five numbers together and `sum()` adds them all; no variable is updated step by step.

Three things stand out in the trace. First, the condition is checked six times but the body runs only five times; the last check is the one that ends the loop. Second, when the loop ends `i` is 6, not 5. Because `i` is declared in the loop header, it cannot be used after the loop anyway, but the detail matters for understanding off-by-one errors. Third, every intermediate value of `sum` is the input of the next pass, so a mistake in one pass is carried into all the later ones.

The two routes cost the same. Both perform five additions; for *n* numbers, *n* additions are needed. The difference is in how the code is written, not in the amount of work. The last panel of Figure 2.1.3 sums up the difference: the two approaches give the same result, and what differs is whether the code says how to do it or what is wanted.

Example 2.1.1

Write the sum of the numbers from 3 to 6 in both ways.

Imperative: `int t = 0; for (int i = 3; i <= 6; i++) t = t + i;`. The passes make `t` equal to 3, 7, 12, and 18 in turn; when `i` is 7 the condition is false.

Functional: `IntStream.rangeClosed(3, 6).sum()`. The stream produces 3, 4, 5, 6, and their sum is 18.

Both routes give 18. Had `range(3, 6)` been written instead of `rangeClosed(3, 6)`, the upper bound would be excluded and the result would be 3 + 4 + 5 = 12.

Common mistake

Forgetting to give the sum variable an initial value. If you write `int sum;` and then `sum = sum + i;` in the loop, Java stops compilation with `variable sum might not have been initialized`. Local variables have no default value. In other languages such a variable may start with a random value and a wrong sum is produced silently; Java catches this error at compile time.

2.1.2.4 Logic programming: Prolog

The best-known example of the logic paradigm is Prolog. A Prolog program consists not of steps but of **facts** and **rules**. The programmer writes what is true; when a question is asked, Prolog searches for the answer itself.

```
% facts
parent(ali, ayse).
parent(ayse, can).
% rule
grandparent(X, Z) :- parent(X, Y),
                     parent(Y, Z).
% query
?- grandparent(ali, can).
true.
```

The first two lines declare two facts: Ali is a parent of Ayşe, and Ayşe is a parent of Can. In Prolog, names that begin with a lowercase letter are constants and those that begin with an uppercase letter are variables. The rule line reads “X is a grandparent of Z if X is a parent of some Y and Y is a parent of Z.” The symbol `:-` means “if,” and the comma means “and.”

The query `?- grandparent(ali, can).` asks “Is Ali a grandparent of Can?” Prolog applies the rule, substituting `ali` for X and `can` for Z. It then looks for a Y that satisfies the fact `parent(ali, Y)` and finds `ayse`. Finally it checks whether the fact `parent(ayse, can)` exists. It does, so the answer is `true`.

None of this search order is written in the program. The programmer built no loop and did not say which fact to try first. In the logic paradigm, control flow is left to the language’s own search method. The same program also answers other questions: the query `?- grandparent(ali, X).` means “Whose grandparent is Ali?” and Prolog finds `X = can`. In an imperative language these two questions would need two separate pieces of code.

2.1.2.5 Object-oriented programming: Rectangle

The object-oriented paradigm is the main subject of this course. The basic idea is to gather related data and the operations that work on that data into a single unit. This unit is called an **object**, and the template for objects is a **class**.

Definition 2.1.7

Object-oriented programming (OOP) builds a program from objects that combine state (data) and behavior (methods). The variables that hold an object’s data are called **fields**, and the operations that define its behavior are called **methods**.

The class below models a rectangle.

```
public class Rectangle {
    double width, height;          // data (state)

    Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    double area() {          // behavior
        return width * height;
    }

    public static void main(String[] args) {
        Rectangle d = new Rectangle(3, 4);
        System.out.println("Area: " + d.area());
    }
}
```

```
Area: 12.0
```

The class has four parts:

- `double width, height;` declares two fields. Each `Rectangle` object carries its own `width` and `height`; these two values are the object's state.
- `Rectangle(double width, double height)` is a **constructor**. A constructor has the same name as the class and no return type. It is called once when an object is created and gives the fields their initial values.
- `double area()` is a method. It computes the area from the object's own fields and returns a `double` value.
- The `main` method is the entry point of the program. It creates an object, calls the object's `area()` method, and prints the result.

The line `this.width = width;` inside the constructor looks odd at first. There are two different `width` names here. The one on the right is the constructor's parameter and carries the value passed in the call. The `this.width` on the left is the object's field. The keyword `this` means "the object currently being worked on." When a parameter has the same name as a field, the parameter shadows the field, and you must write `this.` to reach the field.

In the `area()` method, by contrast, `this` is not written: `return width * height;`. Since the method has no parameter or local variable named `width`, the name goes straight to the field. Writing `return this.width * this.height;` would mean the same thing.

Common mistake

Forgetting `this` in the constructor: `width = width;`. This line compiles, but it assigns the parameter to itself and the field does not change. The object's `width` and `height` fields stay at the default 0.0 and the program prints `Area: 0.0`. The compiler reports no error here because the line is valid as far as the language is concerned. This is a logic error.

2.1.2.6 Code trace: a Rectangle object

When tracing an object-oriented program, two regions of memory are considered together. The **stack** holds the frames of method calls and the local variables. The **heap** is where objects created with `new` live. The local variable `d` is on the stack; the `Rectangle` object it refers to is on the heap.

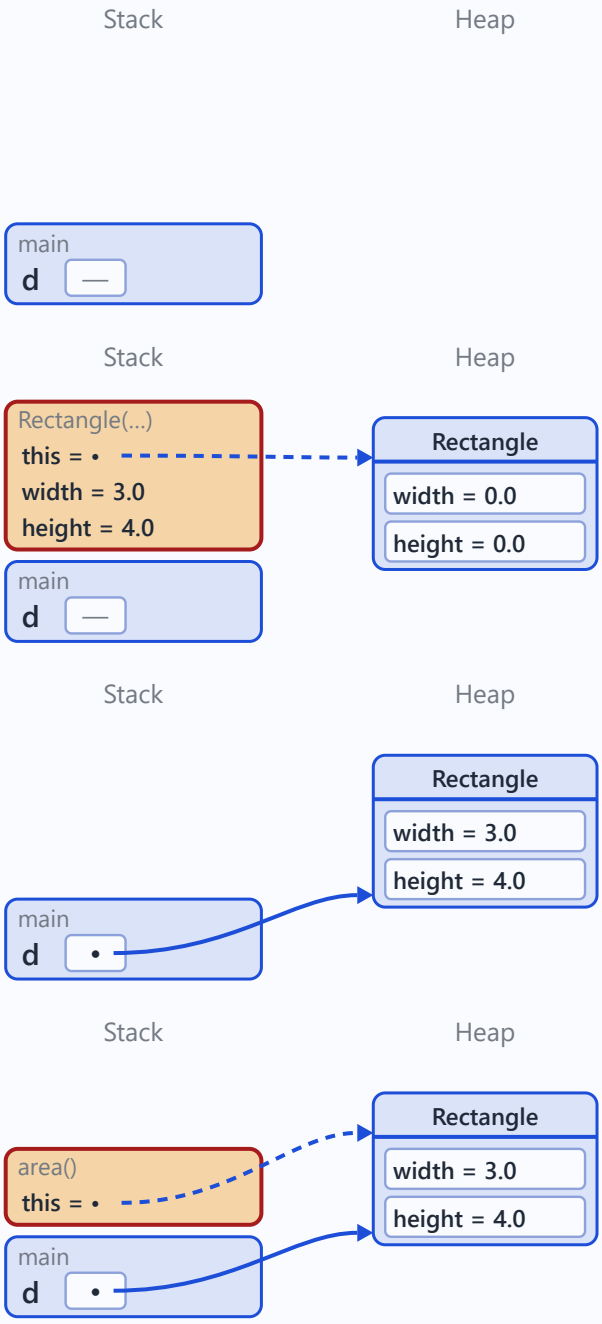


Figure 2.1.4: Four moments from the trace of the Rectangle program. (a) `main` starts; the stack holds the `main` frame, and `d` does not yet refer to an object. (b) `new` allocates the object on the heap and the fields are the default 0.0; in the constructor's frame `this` refers to the object and the parameters are 3.0 and 4.0. (c) The constructor has finished, the fields are 3.0 and 4.0, and `d` now refers to the object. (d) In the call `d.area()`, `this` and `d` refer to the same object.

Figure 2.1.4 gives the flow of the program step by step. The details are as follows:

1. `main` starts running. A `main` frame is opened on the stack and space is set aside for the variable `d`, but there is no object yet.

2. The expression `new Rectangle(3, 4)` is evaluated. `new` allocates space for the object on the heap and gives the fields the default value 0.0. Then the constructor is called. The 3 and 4 in the call are integers while the parameters are `double`, so 3 and 4 are automatically widened to 3.0 and 4.0.
3. `this.width = width;` runs: the object's `width` field becomes 3.0.
4. `this.height = height;` runs: the `height` field becomes 4.0. The constructor ends and its frame is removed from the stack.
5. The value of the `new` expression is the object's **reference**, that is, a value that points to its place on the heap. This reference is assigned to `d`.
6. `d.area()` is called. In the new frame, `this` refers to the same object as `d`. `width * height` = 3.0 × 4.0 = 12.0 is computed and returned.
7. The returned 12.0 is concatenated with the string `"Area: "` and `Area: 12.0` is printed.

The output shows 12.0, not 12, because of the type. `area()` returns a `double`. When Java converts a `double` to a string it writes at least one decimal digit, even if the value is a whole number.

This small program carries the essence of the object-oriented paradigm. Data (`width`, `height`) and behavior (`area`) sit in the same object. The `main` method does not know how the area is computed; it simply asks the object for its area. Even if the method of computation changed, for example if the dimensions were kept in millimeters instead of centimeters, the call in `main` would stay the same.

Example 2.1.2

Add these two lines to `main`: `Rectangle r = new Rectangle(2, 5);` and `System.out.println(r.area() + r.width);`.

A second object is created on the heap; its fields are 2.0 and 5.0. The first object is not affected, because each object has its own fields. `r.area()` returns 10.0 and `r.width` is 2.0. The two `double` values are added and `12.0` is printed.

If the line were `"Area: " + r.area() + r.width`, the result would be different. Because the left operand is a string, both `+` operators concatenate, and `Area: 10.02.0` is printed.

2.1.3 Syntax and semantics

Programming languages and natural languages share a good deal. Both have primitive elements, both have rules for how those elements may be combined, and a correctly formed text has a meaning. Three layers of studying a language rest on this observation: syntax, static semantics, and semantics. The distinction is not merely theoretical. The layer of an error in a program determines who finds it, and when: the compiler, the JVM, or the person testing the program.

2.1.3.1 Primitives and syntax

In English the primitive elements are words and punctuation marks. In Java they are numbers, strings, simple operators (`+ - * /`), and keywords. `int`, `class`, `if`, and `return` are examples of keywords; the language reserves these words for itself.

> **Definition 2.1.8**
>
> **Syntax** is the set of formatting rules that determine which sequences of characters and symbols are well formed in a language. A text that breaks these rules is invalid even if every word in it is correct on its own.

Each word in "cat dog child" is English, but there is no sentence: no subject, object, or predicate has been formed. The same holds in Java. `int x = 5;` is a valid line: first the type, then the name, then the assignment sign, then the value, and finally the semicolon. If the same pieces are arranged in another order, the line is invalid:

```
int = x 5;
```

The compiler rejects this line. The full message from JDK 21 is:

```
Bad.java:3: error: not a statement
        int = x 5;
        ^
Bad.java:3: error: ';' expected
        int = x 5;
           ^
Bad.java:3: error: not a statement
        int = x 5;
              ^
Bad.java:3: error: ';' expected
        int = x 5;
               ^
4 errors
```

A single misordering on a single line has caused four error messages. The reason is that the compiler does not stop after the error; it tries to make sense of the rest of the line. Expecting a name after `int`, it sees `=` and says "this is not a statement." Then it expects a `;` and does not find one. It also tries to interpret the remaining pieces `x` and `5` separately and produces two more messages. The later messages are side effects of the first error. That is why, when several errors are listed, you start fixing with the **first** one; once it is fixed, the others often disappear.

Syntax also affects how understandable a language is. In a language with precise rules, there is no argument about how a line should be read. Syntax errors are therefore the easiest errors to find: the rules are explicitly defined and the compiler checks every line against them.

2.1.3.2 Static semantics

Not every text with correct syntax is meaningful. The English sentence “I are big” fits the pattern `<noun> <verb> <noun>` in form. But “I” is singular and “are” is plural, so the sentence is invalid in the language. This kind of mismatch lies not in the order of the sentence but in how well its parts fit together.

> **Definition 2.1.9**
>
> **Static semantics** are the rules that check, before the program runs, whether a syntactically correct expression is meaningful. In Java the most important of these rules is type compatibility.

In Java, `<value> <operator> <value>` is a valid pattern. `2.3 / 4` fits it and is meaningful. The line below fits the same pattern, but dividing a number by a string has no meaning:

```
double result = 2.3 / "abc";
```

```
error: bad operand types for binary operator '/'
  first type:  double
  second type: String
```

The message says that the `/` operator cannot be used with these two types, and it gives the type of each operand: the first is `double`, the second is `String`. Another error of the same kind is assigning a value to a variable whose type does not match:

```
int x = "five";
```

```
error: incompatible types:
  String cannot be converted to int
```

There is nothing wrong with the syntax of these two lines. The error is found in the compiler’s type-checking step. Java is a **statically typed** language: the compiler knows the type of every variable and every expression and checks the operand types of every operator. That is why errors of this kind are caught before the program ever runs.

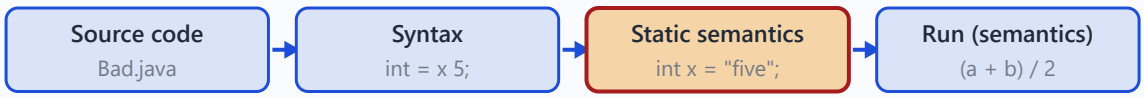


Figure 2.1.5: The checks that source code passes through. The code first goes through the syntax check (`int = x 5;` is stopped here), then the static-semantics check (`int x = "five";` is stopped here); both happen at compile time. Semantic errors (such as an average computed with `(a + b) / 2`) appear only while the program runs.

Figure 2.1.5 shows the three layers as a pipeline. Each layer takes the code that got through the one before it. A line with a syntax error never reaches the type check. Code that passes the type check compiles and can be run. The errors in the last box are of a kind the compiler cannot see.

How strong static-semantics checking is varies from language to language. In languages where this checking is weak, errors that slip through make the behavior of the program unpredictable. In C, writing outside the bounds of an array is not caught at compile time, and the program can silently corrupt another part of memory. Java catches some of these errors at compile time and the rest at run time with an exception; it leaves none of them silent.

2.1.3.3 Semantics

Definition 2.1.10

Semantics determines what a syntactically correct expression means, that is, what it does when it runs.

In natural languages a sentence can have more than one meaning. “They saw the man with the telescope.” can be read in two ways: they used a telescope to see the man, or they saw the man who had the telescope. The listener works out the right meaning from context. Programming languages have no such ambiguity. Every valid expression has exactly one meaning, and that meaning is written in the language specification.

That single meaning, however, may not be what the programmer expects. The code below tries to compute the average of 7 and 8 in three different ways:

```
int a = 7, b = 8;
int avg = (a + b) / 2;
System.out.println(avg);           // not 7.5
System.out.println(a + b / 2);     // precedence
System.out.println((a + b) / 2.0); // correct
```

```
7
11
7.5
```

The average of 7 and 8 is 7.5. Only the third line prints that value. The first two lines compile, run, and print a number, but the number is wrong. The compiler sees no error in these lines because, as far as the language is concerned, there is none. Each line does exactly what is written, according to Java's rules.

- `(a + b) / 2`: both operands are `int`, so Java performs **integer division**. The fractional part of 15 / 2 is discarded and the result is 7.
- `a + b / 2`: there are no parentheses, and `/` has higher precedence than `+`. First `b / 2` = 4 is computed, then 7 + 4 = 11.
- `(a + b) / 2.0`: `2.0` is a `double` constant. 15 is also widened to `double` and real division is performed: 7.5.

2.1.3.4 Code trace: integer division

How an expression is evaluated is shown clearly by its **expression tree**. In the tree every internal node is an operator and every leaf is a value or a variable. An operator can be applied only after the values of its child nodes have been found, so evaluation proceeds from the leaves toward the root.

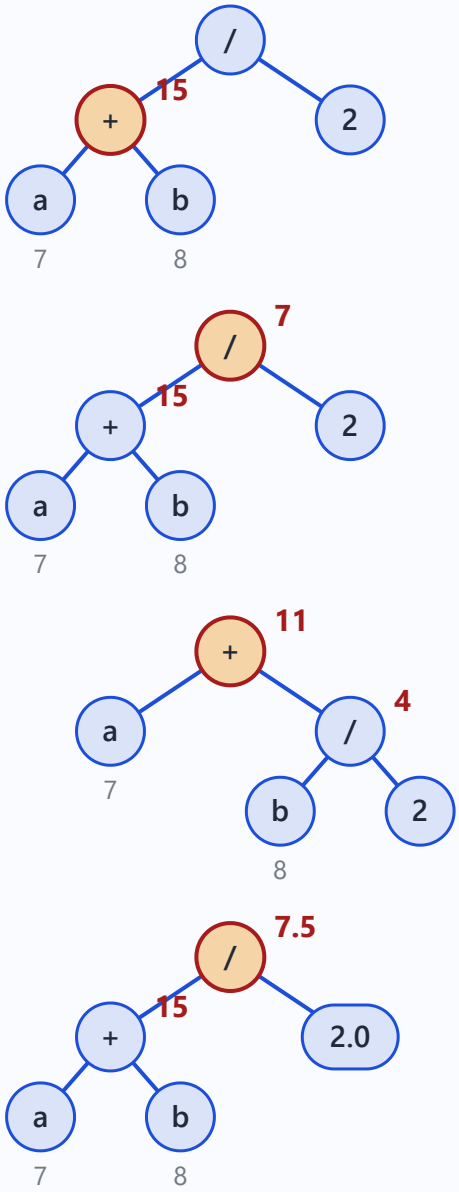


Figure 2.1.6: Expression trees of the average program. (a) `(a + b) / 2`: the parentheses come first, `a + b` = 15. (b) `15 / 2`: both operands are `int`, so integer division gives 7. (c) `a + b / 2`: without parentheses the division sinks to the bottom; first `b / 2` = 4, then 7 + 4 = 11. (d) `(a + b) / 2.0`: because 2.0 is a `double`, the result is 7.5.

Trees (a) and (c) in Figure 2.1.6 consist of the same four pieces (`a` , `b` , `+` , `/`) and the same constant 2. Only their shapes differ. The parentheses move the `+` node down, to the place where it is done first. Without parentheses, the precedence rule puts `/` at the bottom. This is why two expressions written with the same characters give different results.

Line	Computation	Variables	Output
1	—	`a` = 7, `b` = 8	
2	(7 + 8) / 2 = 15 / 2 = 7	`avg` = 7	

Line	Computation	Variables	Output
3	—		7
4	7 + 8 / 2 = 7 + 4 = 11		11
5	(7 + 8) / 2.0 = 15.0 / 2.0 = 7.5		7.5

Table 2.1.5: Line-by-line trace of the average program.

The second line of the program shows the most common trap in these notes. The variable `avg` is declared as `int`. Even if the right-hand side produced 7.5, Java would not allow that value to be assigned to an `int`. Making the variable a `double` does not solve the problem either: the right-hand side still produces 7 by integer division, and the assignment merely turns 7 into 7.0.

Common mistake

Thinking that storing the result in a `double` variable makes the division fractional. The line `double avg = (a + b) / 2;` gives 7.0, not 7.5. Java first evaluates the right-hand side with its own types: the quotient of two `int` values is an `int`. Only after that computation is finished is the assignment performed, and the lost fraction does not come back. The fix is to make one of the operands a `double`: `(a + b) / 2.0`.

2.1.4 Kinds of errors and error messages

Writing programs includes making mistakes and finding them. Even experienced programmers do not write error-free code on the first try. The difference is that they quickly recognize the kind of an error and know where to look for it. The distinction among syntax, static semantics, and semantics pays off directly here.

2.1.4.1 Four kinds of errors

Kind of error	When is it noticed?	Java example	Symptom
Syntax	at compile time (compiler)	`System.out.println("Hello")`	`';' expected`
Static semantics	at compile time (type check)	`int x = "five";`	`incompatible types`
Run-time	while the program runs	`a / b` (when `b` is zero)	crash: `ArithmeticException`
Logic	not by the compiler; only by testing	`(a + b) / 2` for an average	wrong result, infinite loop

Table 2.1.6: Kinds of errors, when they are noticed, and how they show up in Java.

The first two kinds are together called **compile-time errors**. Both are found before the program runs, and no `.class` file is produced while one is present. Syntax errors are like spelling mistakes: they violate clearly defined rules, and the compiler finds them easily. Static-semantics errors are looked for with language-specific checks; in Java the types are at the center of this checking.

Definition 2.1.11

A **run-time error** arises while a compiled program is running and, in Java, causes an **exception**. A **logic error** is when the program runs and produces a wrong result; neither the compiler nor the JVM notices it.

The last row of the table describes the most dangerous case. Even a program that is flawless in syntax and static semantics may not produce the expected result. It can crash, fall into an infinite loop, or give a wrong result without any warning. The last of these is the hardest to find, because nothing signals that something has gone wrong.

2.1.4.2 Reading a compile-time error message

A compiler's error message can look frightening, but it has a regular structure. In the program below a semicolon is missing:

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello")
    }
}
```

```
Hello.java:3: error: ';' expected
        System.out.println("Hello")
                                   ^
1 error
```

The message has four parts:

- `Hello.java:3` : the file and line number where the error was found.
- `error: ';' expected` : what the compiler was expecting. Here a semicolon was expected.
- A copy of the line with a `^` mark under it: the position of the error within the line. The mark points just past the closing parenthesis, where the `;` should have been.
- `1 error` : the total number of errors.

When there is an error, no `.class` file is produced and the program does not run at all. If an old `.class` file is left over from an earlier compilation, the `java` command may run that old version by mistake. This is often why a change to the code seems to have no effect: the compilation failed, but the message went unnoticed.

When a semicolon is missing, `javac` points to the error at the end of the line where the `;` is missing. In the fragment below, the `;` at the end of the first line has been forgotten:

```
int x = 5
int y = 6;
System.out.println(x + y);
```

```
Semi.java:3: error: ';' expected
        int x = 5
                 ^
1 error
```

The mark is placed not on the second line but right after the `5`. When the compiler sees the next token after `5` (`int`), it realizes the statement should have ended and reports the omission at the end of the previous piece. Line 3 of the file is the first line of the fragment; above the fragment are the class and `main` headers.

Example 2.1.3

A program declares `int total = 0;` and on line 7 writes `System.out.println(Total);`. The compiler reports `Calc.java:7: error: cannot find symbol`, with `symbol: variable Total` beneath it.

Reading the message: the error is on line 7, and the compiler could not find a variable named `Total`. The declared name is `total`. Java distinguishes uppercase from lowercase, so these are two different names. This is a compile-time (static semantics) error. When `Total` is changed to `total`, the program compiles.

2.1.4.3 Run-time errors and exceptions

The program below compiles without trouble. The error appears only when it runs:

```
public class Divide {
    public static void main(String[] args) {
        int a = 10, b = 0;
        System.out.println(a / b);
    }
}
```

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at Divide.main(Divide.java:4)
```

The compiler sees nothing wrong with `a / b`: the quotient of two `int` values is a valid expression. That `b` is 0 at that moment is known only while the program runs. Integers cannot be divided by zero, so the JVM throws an **exception**. The parts of the message are:

- `Exception in thread "main"`: the exception occurred in the `main` thread.
- `java.lang.ArithmeticException`: the type of the exception, that is, the full name of its class.
- `/ by zero`: the description of the exception; an integer division by zero was attempted.
- `at Divide.main(Divide.java:4)`: the method, file, and line where the exception occurred.

In the real output the first two parts are printed on one line; here they are split to fit the page. If the exception is not caught by a `try`-`catch` block, the program ends at this line and the following lines never run.

The behavior of division by zero depends on the type. `10 / 0` throws an exception. `10.0 / 0` does not throw; it produces the value `Infinity`, and `0.0 / 0` gives `NaN` (not a number). Floating-point numbers have these special values; integers do not. A division by zero with `double` therefore does not crash the program, but silently turns the result into a meaningless value.

Another common run-time error is going outside the bounds of an array. `int[] a = new int[3];` creates an array of three elements; the valid indices are 0, 1, and 2. `a[3] = 1;` compiles, but at run time it throws `ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3`. The value of an index is often the result of a computation, and the compiler cannot know it in advance.

2.1.4.4 Logic errors and testing

With a logic error the program compiles, runs, and gives a result; only the result is wrong. Computing the average as `(a + b) / 2`, writing `2 * pi * r` for the area of a circle (that is the circumference), and setting the bound of a loop one too low or one too high are all of this kind. As far as the language is concerned, none of these lines is flawed.

The way to find logic errors is **testing**: run the program on inputs whose result is known in advance and compare the output with the expected value. The average of 7 and 8 is known to be 7.5; if the program prints 7, there is a bug. A few well-chosen tests also narrow down where the bug is. For 6 and 8 the program gives the right result (7), and for 7 and 8 the wrong one; this difference suggests that fractional results are being lost.

The exception mechanism does not catch logic errors. An exception is thrown when an invalid operation is performed; with a logic error every operation performed is valid. The distinction matters: that a program runs without throwing an exception does not show that it runs correctly.

Example 2.1.4

Suppose each of the four lines below is in a separate program. Identify the kind of error.

(a) `String s = 5;` A static-semantics error. The compiler says `incompatible types: int cannot be converted to String`. The form of the line is correct; the types do not match.

(b) `System.out.println("Hello";` A syntax error. The closing parenthesis is missing; the compiler says `')' or ',' expected`.

(c) `int k = 0; System.out.println(10 % k);` A run-time error. Like division, the remainder operation cannot be taken with a zero divisor; `ArithmeticException: / by zero` is thrown.

(d) `double perimeter = pi * r * r;` A logic error. The code compiles and runs, but it computes the area instead of the perimeter. The correct form is `2 * pi * r`.

2.1.5 Expressions and operator precedence

Every part of a program that does a computation is an expression. To find the value of an expression correctly, two things must be known: the order in which the operations are carried out, and the type each operation produces. This section covers the first, the precedence rules; types come into those rules here and there.

2.1.5.1 Expressions, operators, operands

Definition 2.1.12

An **expression** is a construct that produces a value when evaluated: `3 + 4`, `x > 0`, `pi * r * r`. The operation symbols in an expression are called **operators**, and the values an operator works on are called **operands**.

In the expression `3 + 4`, `+` is the operator, 3 and 4 are the operands, and the value of the expression is 7. Operands can themselves be expressions. In `(a + b) / 2`, the left operand of the `/` operator is the expression `(a + b)`. Expressions thus nest inside one another and form a tree.

There are five basic arithmetic operators in Java: addition `+`, subtraction `-`, multiplication `*`, division `/`, and remainder `%`. The remainder operator gives the remainder of a division: `7 % 3` = 1, because 7 = 2 · 3 + 1. In addition, the comparison operators (`<`, `>`, `<=`, `>=`, `==`, `!=`) compare two values and produce a `boolean` value, that is, `true` or `false`.

2.1.5.2 Precedence and the left-to-right rule

Definition 2.1.13

Operator precedence is the rule that determines the order in which operators are applied in an expression. Arithmetic operators of the same precedence are applied **from left to right**.

Java's arithmetic precedence matches the rules of school mathematics. Table 2.1.7 lists the operators used in these notes from highest precedence to lowest.

Level	Operators	Direction
1	parentheses `( )`	inside out
2	`*` `/` `%`	left to right
3	`+` `-` (addition, subtraction, string concatenation)	left to right
4	`<` `>` `<=` `>=`	left to right
5	`==` `!=`	left to right
6	`=` (assignment)	right to left

Table 2.1.7: Precedence of the operators used in these notes (decreasing from top to bottom).

Two details in the table are easy to miss. First, `*`, `/`, and `%` are on the same level. None comes before the others; whichever is further left is done first. The expression `12 / 3 * 2` gives (12 / 3) · 2 = 8, not 12 / 6 = 2. Second, assignment has the lowest precedence and works right to left. So `x = y = 7;` first assigns 7 to `y` and then assigns the value of that assignment (7) to `x`.

Performing operations of equal precedence from left to right directly affects the result for subtraction. The expression `10 - 4 - 3` is computed as (10 − 4) − 3 = 3. Starting from the right would give 10 − (4 − 3) = 9. For addition and multiplication the order does not change the result, but once integer division is involved it does: `7 / 2 * 2` = 3 · 2 = 6, whereas `7 * 2 / 2` = 14 / 2 = 7.

Example 2.1.5

Find the value of the expression `10 - 2 * 3 + 8 / 4 % 3`.

The operators of highest precedence are `*`, `/`, and `%`. From left to right: `2 * 3` = 6. Then `8 / 4` = 2, and then `2 % 3` = 2. The expression becomes `10 - 6 + 2`.

`-` and `+` are on the same level, left to right: 10 − 6 = 4, 4 + 2 = 6. The result is **6**, and its type is `int`.

2.1.5.3 Evaluating an expression step by step

A computer does not evaluate an expression in one go; it simplifies it step by step according to the precedence rules. At each step it performs the leftmost of the operations with the highest precedence and puts its result in its place. The following expression, taken from the Fahrenheit-to-Celsius formula, shows this well: `(72 - 32) / 9 * 5`. In mathematics its value is 40 · 5 / 9 ≈ 22.22.

Step	Expression	Operation performed
1	`(72 - 32) / 9 * 5`	parentheses first: 72 − 32 = 40
2	`40 / 9 * 5`	`/` and `*` are on the same level, left to right: first 40 / 9
3	`4 * 5`	both operands are `int`: integer division gives 4, not 4.44… (the remainder 4 is discarded)
4	`20`	the result is 20; the expected value was 22.22…
5	`(72.0 - 32) / 9 * 5`	`double` − `int`: 32 is widened to 32.0; the result is 40.0
6	`40.0 / 9 * 5`	real division: 4.444444444444445
7	`4.444444444444445 * 5`	`double` × `int` → `double`
8	`22.22222222222222`	the correct result

Table 2.1.8: Step-by-step evaluation of the expressions `(72 - 32) / 9 * 5` and `(72.0 - 32) / 9 * 5`.

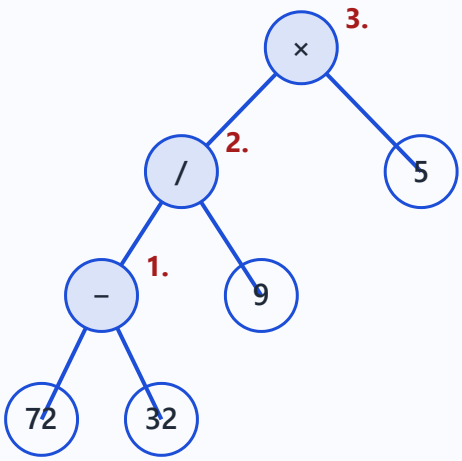


Figure 2.1.7: The tree of the expression `(72 - 32) / 9 * 5`. The operation at the bottom is done first; the numbers give the order of operations: 1. subtraction, 2. division, 3. multiplication.

Figure 2.1.7 shows the structure of the expression. The root is the multiplication, because it is done last. The left operand of the multiplication is the division, and the left operand of the division is the subtraction. The left-to-right rule shows up in the tree like this: since `/` and `*` have the same precedence, the one on the left (`/`) sinks to the bottom and is done first.

The first four steps of Table 2.1.8 show that a result can come out wrong even when the precedence rules are applied correctly. The error is not in the precedence but in the type. In the third step two `int` values are divided and the fraction is lost. That loss is magnified by the multiplication that follows: the lost 0.44 is multiplied by five, and the result is 2.22 too small. This is a semantic (logic) error; the compiler does not warn about it.

To fix it, it is enough to make a single constant a `double`. When `72.0` is written, the result of the subtraction becomes `double`, and this type spreads through the rest of the expression. If one operand of an operation is `double`, the other is widened to `double` as well and the result is `double`. If a fractional result is needed, at least one operand must be `double`, and this widening must happen *before* the division.

The last digit of 22.22222222222222 in the final step may also draw attention. In mathematics 200 / 9 = 22.222… goes on forever. A `double` keeps about 15–17 significant digits; when Java prints the value, it chooses the shortest digit string that identifies it uniquely. Floating-point numbers are therefore always approximations.

Example 2.1.6

Write the same computation with the order of operations changed: `(72 - 32) * 5 / 9`.

Parentheses: 40. Left to right, multiplication first: 40 · 5 = 200. Then division: 200 / 9 = 22 (integer division; the remainder 2 is discarded). The result is **22**.

The loss is smaller: because the integer division is left for last, only the final fraction (0.22…) is discarded. Doing the multiplication before the division reduces the error when working with integers, but does not remove it. For the exact result a `double` is still needed: `200 / 9.0` = 22.22222222222222.

2.1.5.4 Integer division and remainder

Definition 2.1.14

Integer division discards the fractional part when one integer is divided by another. In Java, if both operands of the `/` operator are of an integer type (such as `int` or `long`), the result is an integer too. The result is truncated toward zero: `7 / 2` = 3, `-7 / 2` = −3.

The phrase “truncated toward zero” matters for negative numbers. −7 / 2 = −3.5. When the fraction is discarded, −3 remains; the result moves toward zero. Some languages (for example Python’s `//` operator) round down and give −4. Java always truncates toward zero.

The remainder operator `%` is consistent with division. For every `a` and every nonzero `b`, the following identity holds:

> (a / b) * b + a % b == a

It follows from this identity that the sign of the remainder is the same as the sign of the dividend. `-7 % 2` = −1, because (−3) · 2 + (−1) = −7. `-17 % 5` = −2. By contrast, `7 % -2` = 1; the sign of the divisor does not affect the remainder. Whether a number is even is tested with `n % 2 == 0`. Because the remainder of a negative odd number is −1, an odd-number test must be written `n % 2 != 0`, not `n % 2 == 1`.

Expression	Result	Type	Explanation
`7 % 3`	`1`	`int`	7 = 2 · 3 + 1
`17 / 5`	`3`	`int`	the fraction of 3.4 is discarded
`17 % 5`	`2`	`int`	17 = 3 · 5 + 2
`-7 / 2`	`-3`	`int`	truncated toward zero
`-7 % 2`	`-1`	`int`	the remainder takes the sign of the dividend
`1 / 3 * 3`	`0`	`int`	1 / 3 = 0, then 0 · 3
`1.0 / 3 * 3`	`1.0`	`double`	real division
`17 / 5.0`	`3.4`	`double`	one operand is `double`

Table 2.1.9: Examples of integer division and remainder (verified by running them with JDK 21).

Integer division is not always a mistake. Many algorithms use it on purpose. Splitting seconds into minutes and seconds is a typical example: for 135 seconds, `135 / 60` = 2 minutes and `135 % 60` = 15 seconds. Separating the digits of a number works the same way: `n % 10` gives the last digit and `n / 10` drops it. The problem arises when integer division happens unnoticed where a fractional result was expected.

Example 2.1.7

A student’s three exam scores are 70, 85, and 90. Compute the average in three ways:

```
int t1 = 70, t2 = 85, t3 = 90;
System.out.println((t1 + t2 + t3) / 3);     // 81
System.out.println((t1 + t2 + t3) / 3.0);   // 81.66666666666667
double avg = (t1 + t2 + t3) / 3;
System.out.println(avg);                     // 81.0
```

The total is 245. The first line finds 245 / 3 by integer division: 81. In the second line, 3.0 is a `double`, so real division is performed and 81.66666666666667 is printed. The third line is the most misleading: even though the variable is `double`, the right-hand side is first computed as 81, and the assignment turns it into 81.0.

2.1.5.5 Exponentiation

Java has no exponentiation operator. The operation written 2^3 in mathematics is done with the method `Math.pow(2, 3)`, and the result is a `double`: `8.0`. `Math.pow(2, 10)` = 1024.0 and `Math.pow(2, 0.5)` = 1.4142135623730951. For small integer powers, multiplication is also used: `x * x` is x^2.

Two habits from other languages cause trouble here. Python's `**` operator does not exist in Java; if `2 ** 3` is written, the compiler says `illegal start of expression`. The more insidious one is the `^` sign. In Java `^` is a valid operator, but it is not exponentiation; it is the bitwise exclusive or (XOR):

```
System.out.println(2 ^ 3);           // 1: XOR
System.out.println(Math.pow(2, 3));  // 8.0
```

The binary representation of 2 is 010 and that of 3 is 011. XOR produces 1 in the bits where the two numbers differ: 001, that is, 1. The code compiles and runs; only the result is wrong. Because it gives no compile-time error, trying to exponentiate with `^` is a logic error.

2.1.6 Variables, assignment, and typing

The values a program computes must be stored somewhere. That place is a variable. The type of a variable determines what kind of value it can hold and what operations can be performed on that value.

2.1.6.1 Variables and assignment

Definition 2.1.15

A **variable** is a memory location bound to a name, whose value can change. In Java a variable is **declared** with its type and name before it is used: `int count;`. **Assignment** writes the value of the expression on the right into the variable on the left, using the `=` sign.

Assignment takes place in two steps, and the order of the two steps matters:

1. The expression on the right is computed with the variables' *current* values.

2. The value found is written into the variable on the left; the variable's old value is erased.

This order makes sense of the line `x = x + 1`, which looks meaningless in mathematics, where it is an equation with no solution. In Java it is a command: "add 1 to the current value of `x` and write the result into `x`." If `x` is 5, first 5 + 1 = 6 is computed, then 6 is written.

Example 2.1.8

What are `a` and `b` after these four lines run? `int a = 4; int b = a * 3; a = b - a; b = b / a;`

1. `a` = 4. 2. The right-hand side is 4 · 3 = 12, so `b` = 12. 3. The right-hand side is computed with the old value of `a`: 12 − 4 = 8; then `a` becomes 8. 4. 12 / 8 = 1 (integer division; 0.5 is discarded); `b` = 1.

Result: `a` = 8, `b` = 1. In the third line `a` appears on both the right and the left. While the `a` on the right is being read, its value is still 4; the new value is written only after the computation is finished.

Variable names are called **identifiers**, and they must follow a few rules. A name begins with a letter, an underscore (`_`), or a dollar sign (`$`); the following characters may also be digits. A name cannot contain spaces and cannot be a keyword. `total2`, `_counter`, `$price`, and `averageGrade` are valid. `2total` is invalid because it starts with a digit, `student count` because it contains a space, and `int` because it is a keyword. By Java convention, variable names start with a lowercase letter and later words are joined with an initial capital: `averageGrade`, `studentCount`.

Java distinguishes uppercase from lowercase. `Count` and `count` are two different names:

```
int Count = 4;
System.out.println(count);
```

```
Case.java:4: error: cannot find symbol
        System.out.println(count);
                           ^
  symbol:   variable count
  location: class Case
1 error
```

The message says that the compiler could not find a variable named `count`. The `symbol` line gives the name being looked up, and the `location` line gives the class in which the lookup was made. The same message appears when the variable was never declared or when its name is misspelled.

A local variable cannot be read before it is given a value:

```
int s;
System.out.println(s);
```

```
Uninit.java:4: error: variable s might not have been initialized
        System.out.println(s);
                           ^
1 error
```

The compiler examines every path that reaches `s` and finds no assignment on any of them. This check prevents a random memory value from being used silently. The fields of an object (such as `width` and `height` of `Rectangle`), by contrast, automatically receive a default value: 0 for numbers and `false` for `boolean`. Local variables have no such default.

2.1.6.2 Assignment versus comparison

In Java `=` is assignment and `==` is comparison. Because the equals sign in mathematics is used in both senses, the two operators are often confused. Writing `=` inside a condition by mistake is a typical error:

```
int x = 100;
if (x = 100) {
    System.out.println("hundred");
}
```

```
Check.java:4: error: incompatible types: int cannot be converted to boolean
        if (x = 100) {
              ^
1 error
```

A `boolean` value is expected inside the parentheses of an `if` statement. `x = 100` is an assignment, and its value is the value assigned, that is, the `int` 100. Java does not convert an `int` to a `boolean`, so the line does not compile. The correct form is `if (x == 100)`. In C, integers can be used as conditions, so the same mistake compiles and the condition is always true; Java's type checking catches this error at compile time.

The values produced by comparison operators can be stored in `boolean` variables: `boolean passed = score >= 50;`. In this line the comparison `score >= 50` is made first (in the precedence table `>=` is higher than `=`), and then its result is assigned to `passed`. Such a variable can be used directly in an `if` statement: `if (passed)`.

2.1.6.3 Static and dynamic typing

Every programming language is concerned with the types of values; the difference between languages lies in when the types are checked.

> **Definition 2.1.16**
>
> In **statically typed** languages (Java, C, C#) the type of a variable is stated explicitly, does not change, and type compatibility is checked at compile time. In **dynamically typed** languages (Python, JavaScript) the type of a variable is not declared in advance; it is determined when a value is assigned and may change at run time. These two approaches are called **static typing** and **dynamic typing**.

The difference shows when the same idea is written in two languages. In Python a variable can be assigned a number first and then a string:

```
x = 5          # x is an int now
x = "five"      # now a str: no problem
print(x + 1)   # at run time:
# TypeError: can only concatenate
#   str (not "int") to str
```

The first two lines run without trouble. The error appears only when the third line is executed and a number is added to a string. If that line were at the end of the program or in a rarely taken branch, the error might go unnoticed for weeks. In Java the same idea does not compile:

```
int x = 5;
x = "five";     // does not compile:
// error: incompatible types:
//   String cannot be converted to int
```

`x` is declared as `int`, and that type does not change. The assignment on the second line is rejected at compile time, and the program never runs.

	Dynamically typed (Python, JavaScript)	Statically typed (Java, C, C#)
Type declaration	Not declared in advance; determined when a value is assigned	Stated explicitly and does not change
Can the type change?	Yes, at run time	No
Incompatible operation	Error at run time (`TypeError`)	Error at compile time

	Dynamically typed (Python, JavaScript)	Statically typed (Java, C, C#)
Benefit	Flexibility, short code	Safety; the compiler can generate fast code

Table 2.1.10: Comparison of dynamic and static typing.

Static typing also brings speed. Because the compiler knows that both variables in `a + b` are `int`, it generates an integer-addition instruction directly. In a dynamically typed language, the current types of the operands must be looked at on every addition: two numbers, two strings, or an incompatible pair?

Since Java 10, local variables can also be declared with `var`. This does not make Java dynamically typed:

```
var count = 3;        // type: int
var ratio = count / 2;  // type: int, value 1
var name = "Ali";       // type: String
count = "three";       // does not compile: String cannot be converted to int
```

With `var` no type is written, but the compiler infers the type from the right-hand side and fixes it at compile time. The type of `count` is `int`; a string cannot be assigned to it later. `var` only shortens the notation; type checking continues exactly as before. For this reason a variable declared with `var` must be given an initial value: `var x;` does not compile, because the compiler has no value from which to infer the type.

Common mistake

Thinking that `var` lets a variable take a value of any type. In the line `var ratio = count / 2;`, if `count` is 3, the type of `ratio` is `int` and its value is 1. `var` does not choose a fractional type; it takes whatever the right-hand side produces. If a fractional result is wanted, either write `double ratio = count / 2.0;` or arrange the right-hand side so that it produces a `double`.

2.1.7 Strings and types

A program does not work with numbers alone. Every message shown to the user, every name read in, and every result written to the screen is a string. When numbers and strings meet in the same expression, Java's rules are precise but surprising at first sight. This section covers those rules.

Definition 2.1.17

A **string** is a piece of text made up of characters. In Java strings are objects of the `String` class and are written in double quotes in source code: `"Hello"`. A single character in single quotes (`'a'`) is not a string but a value of type `char`.

2.1.7.1 Different types cannot be compared

Comparing different data types directly leads to an error. An integer and a string cannot be compared; although `5` and `"5"` look the same to a human eye, one is a number and the other is a sequence of characters.

```
int num = 5;
String text = "5";
boolean b = num == text;
```

```
error: bad operand types for binary operator '=='
  first type:  int
  second type: String
```

Languages define rules and **type conversion** routes for comparing types. A comparison is meaningful only after both values have been brought to the same type. This can go two ways: the string is converted to a number, or the number is converted to a string.

```
int num = 5;
String text = "5";
System.out.println(Integer.parseInt(text) == num);
System.out.println(text.equals(String.valueOf(num)));
System.out.println(text.equals(num));
```

```
true
true
false
```

- `Integer.parseInt(text)` converts the string `"5"` to the `int` 5. The two `int` values can now be compared with `==` : `true`.
- `String.valueOf(num)` converts the number 5 to the string `"5"`. The two strings are compared with the `equals` method: `true`.

- `text.equals(num)` compiles, but always returns `false`. `equals` accepts an object of any type as its parameter; here `num` is automatically wrapped in an `Integer` object. A `String` is never equal to an `Integer`.

The third line is the most dangerous, because the compiler does not warn. The program runs and gives a wrong answer; this is a logic error. `parseInt` also needs care: if the string is not an integer, for example `"5a"` or `"5.0"`, it throws `NumberFormatException` at run time.

Comparing strings has one more rule: strings are compared with `equals`, not with `==`. The operator `==` checks whether two references point to the same object; `equals` checks whether the characters of the two strings are the same. Two separate objects can hold the same text, and then `==` gives `false` while `equals` gives `true`. Because identical string literals in source code are stored as a single object in Java, `==` sometimes gives the right answer by accident. Do not rely on that coincidence.

2.1.7.2 Operations across types

The same operator behaves differently on different types. The clearest example is `+`: on numbers it is addition, and on strings it is **concatenation**.

Definition 2.1.18

String concatenation joins strings end to end with `+`. If one operand of `+` is a string, the other is converted to a string too, and the result is a string. A **widening conversion** is the automatic conversion of a smaller numeric type to a larger one: `int` → `double`.

Expression	Java result	Type	Explanation
`3.0 + 4.0`	`7.0`	`double`	two floating-point numbers
`3 + 4`	`7`	`int`	two integers
`3 + 4.0`	`7.0`	`double`	the `int` is automatically widened to `double`
`"3" + "4"`	`"34"`	`String`	concatenation
`3 + "4"`	`"34"`	`String`	if one side is a string, the other is converted to a string; not an error
`3 + true`	does not compile	—	`bad operand types for binary operator '+'`
`'a' + 1`	`98`	`int`	a `char` is a numeric code (`'a'` = 97)

Table 2.1.11: The `+` operation on different types (verified by running them with JDK 21).

Each row of the table shows a rule. The sum of two `int` values is an `int`. When an `int` and a `double` are added, the `int` is first widened to `double`; no information is lost in widening, because every `int` value can be represented exactly in a `double`. The reverse direction is not automatic. The

line `int n = 4.5;` does not compile, and the compiler says `possible lossy conversion from double to int`: converting a `double` to an `int` can lose the fraction, and Java expects the programmer to ask for that loss explicitly.

The line `3 + "4"` shows the different choices languages make. Java does not count it as an error; it converts 3 to `"3"` and concatenates. In Python, `3 + "4"` throws a `TypeError`; Python does not convert a number to a string on its own. JavaScript concatenates, like Java. None of them is wrong; each language has its own rule, and a programmer must know the rule of the language in use.

The line `3 + true` is another choice Java makes: a `boolean` is not a number. In C, true counts as 1 and this sum gives 4; Java does not compile the line. The last row, `'a' + 1`, looks surprising but follows the rule. The `char` type stores a character as its Unicode code number; the code of `'a'` is 97. In an arithmetic operation the `char` is widened to `int`, and 97 + 1 = 98. To see the result as a character, an explicit cast is needed: the expression `(char) ('a' + 1)` gives `'b'`.

An operation across types leads to unexpected results or errors if the rules of the language are not known. In Java these rules are exact, and the compiler applies them to every expression in the same way. For a programmer who knows the rules, Java's behavior is always predictable.

2.1.7.3 Concatenation goes left to right

The `+` operator works from left to right and at each step looks only at the two operands of that step. When an expression contains both numbers and strings, the result depends on where the string is:

Expression	Steps	Output
`"Total: " + 2 + 3`	`"Total: 2"` + 3	`Total: 23`
`"Total: " + (2 + 3)`	parentheses first: 5; `"Total: " + 5`	`Total: 5`
`2 + 3 + " total"`	2 + 3 = 5; `5 + " total"`	`5 total`
`1 + 2 + "3" + 4 + 5`	3; `"33"`; `"334"`; `"3345"`	`3345`
`"" + 'A' + 'B'`	`"A"`; `"AB"`	`AB`
`'A' + 'B'`	65 + 66	`131`

Table 2.1.12: Expressions mixing numbers and strings.

The first row is the source of a common mistake. The programmer wants to print "Total: 5", but Java starts from the left: `"Total: " + 2` is a string. Adding 3 to this string is again concatenation, and "Total: 23" comes out. Parentheses are needed so that the numbers are added first. In the third row the string comes last. The two `int` values on the left are added to give 5; concatenation happens only in the final step.

The fourth row shows the general form of the rule. Numbers are added until the first string is seen; after the first string, every `+` is concatenation. The last two rows apply the same rule to `char`. In the expression `'A' + 'B'` there is no string; the two characters are added as numbers, and 65 + 66 =

131. Putting an empty string (`""`) at the start turns the whole expression into concatenation from the beginning.

2.1.7.4 The decimal separator in input

Another aspect of conversion between types is input read from the user. Everything that comes from the keyboard is text first; the `Scanner` class converts this text to the requested type. The `nextDouble()` method uses the computer's **locale** when reading a floating-point number. In the Turkish locale the decimal separator is a comma.

```
Scanner input = new Scanner(System.in);   // locale: Turkish
double height = input.nextDouble();
// input "1,75" -> 1.75
// input "1.75" -> java.util.InputMismatchException
```

On a computer with a Turkish regional setting, if the user types `1,75`, the value is read as 1.75. If the user types `1.75`, `Scanner` does not recognize it as a floating-point number and throws `InputMismatchException`; the program stops. Constants in the source code (such as `65.2`) and the output of `println` are not affected by the locale; they always use a period. So the same program asks for a comma in the input and prints a period in the output.

There are two ways to solve the problem. The first is to match the example in the prompt shown to the user to the locale: "Height (e.g. 1,75)". The second is to fix the locale of the `Scanner`: after the line `input.useLocale(Locale.US);` the input is read with a period on every computer. This detail is a common reason why the same program works on one computer and crashes on another.

2.1.8 From algorithm to program

The previous sections took up the building blocks of a program one at a time: values, types, expressions, variables. This section puts them together. A program is a sequence of instructions built from these blocks, and the order of the instructions matters at least as much as the instructions themselves.

2.1.8.1 A program is a sequence of instructions

A computer processes instructions one at a time, in order. When one instruction finishes, the next one begins, and each instruction continues from the state the previous one left behind. That is why it is important to get the order of operations right. A classic example is exchanging the values of two variables, which is called a **swap**.

```
int a = 3, b = 5;
a = b;
b = a;
System.out.println(a + " " + b);

int c = 3, d = 5;
int temp = c;
c = d;
d = temp;
System.out.println(c + " " + d);
```

The program prints two lines: `5 5` and `5 3`. The first swap fails and the second succeeds. Table 2.1.13 traces the two fragments line by line.

Line	Statement	Variables	Output
1	`int a = 3, b = 5;`	`a` = 3, `b` = 5	
2	`a = b;`	`a` = 5, `b` = 5 (the old 3 is gone)	
3	`b = a;`	`a` = 5, `b` = 5	
4	`println`		`5 5`
6	`int c = 3, d = 5;`	`c` = 3, `d` = 5	
7	`int temp = c;`	`c` = 3, `d` = 5, `temp` = 3	
8	`c = d;`	`c` = 5, `d` = 5, `temp` = 3	
9	`d = temp;`	`c` = 5, `d` = 3, `temp` = 3	
10	`println`		`5 3`

Table 2.1.13: Trace of the swap program.

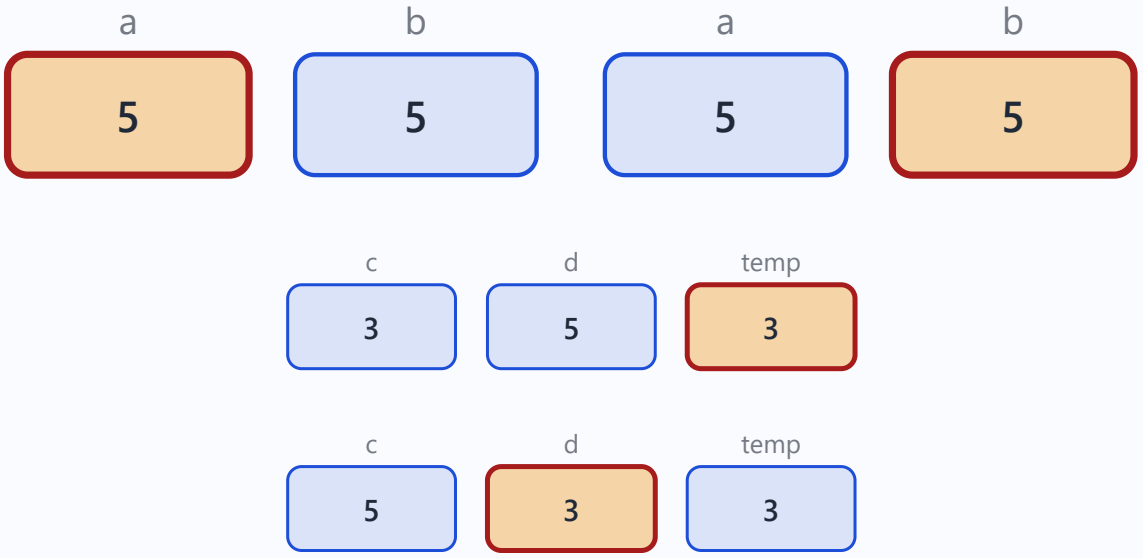


Figure 2.1.8: Four moments from the swap trace. (a) `a = b`: 5 is written into `a` and the old value 3 is lost. (b) `b = a`: the current value of `a` (5) is written into `b`; the swap has failed. (c) The correct method: the value of `c` is first saved in `temp`. (d) The saved old value of `c` (3) is written into `d`; the swap succeeds.

The flaw in the first fragment is that assignment erases the old value. When `a = b;` runs, the value 3 of `a` is lost and has been saved nowhere. The following line `b = a;` copies the current value of `a`, which is 5. Panels (a) and (b) of Figure 2.1.8 show this loss. Reordering the lines does not fix the result: if `b = a;` is written first, this time the value of `b` is lost and the output is `3 3`.

The correct method is to save the value that is about to be overwritten in a third, **temporary** variable beforehand. The line `temp = c` puts the 3 in safekeeping. After that `c` can be overwritten safely. The last line moves the saved value into `d`. A swap takes three assignments, and their order cannot be changed. The cost is constant: however many values there are, one swap needs three assignments and one extra variable.

Example 2.1.9

Integers can also be swapped without a temporary variable. For `a` = 4 and `b` = 9, trace these three lines: `a = a + b; b = a - b; a = a - b;`

1. `a` = 4 + 9 = 13, `b` = 9. 2. `b` = 13 − 9 = 4. 3. `a` = 13 − 4 = 9.

In the end `a` = 9 and `b` = 4. The method works, but it has two drawbacks. It is hard to read, and if the sum `a + b` exceeds the range of `int`, overflow occurs. The method with a temporary variable works for every type and shows its intent clearly.

2.1.8.2 Expressions, statements, and programs

The parts of a program are of two kinds: those that produce a value and those that produce an effect.

Definition 2.1.19

An expression computes a value and returns the result: the value of the expression `3 + 4` is 7. A **statement** produces an effect: `pi = 3.14159;` assigns a value to a variable, and `System.out.println(pi);` writes to the screen. In Java statements usually end with `;`. A program consists of statements that run in order.

Expressions can nest and can appear inside statements. `double area = pi * r * r;` is a statement; it contains the expression `pi * r * r`, which in turn contains the expression `pi * r`. The reverse does not hold: a statement cannot appear inside an expression. `println` writes to the screen but returns no value; its return type is `void`. Using it inside a computation therefore makes no sense:

```
double d = 3 + System.out.println(pi);
```

```
error: 'void' type not allowed here
```

Conversely, an expression that produces a value but has no effect cannot stand alone as a statement:

```
3 + 4;      // produces a value, no effect
```

```
error: not a statement
```

An expression does not become a statement just by putting `;` after it. Java allows this for only four kinds of expression: assignment (`x = 5;`), increment and decrement (`i++;`), method call (`println(x);`), and object creation (`new Rectangle(1, 2);`). What they have in common is that they produce an effect. `3 + 4;`, `x * 2;`, or `x > 3;` is computed and the result is immediately thrown away; this is wasted work, and the compiler treats it as a typing mistake.

There is a subtle point here. In Java an assignment also produces a value: the value assigned. That is why the following line compiles:

`int y = (x = 4) + 1;` → `x` = 4, `y` = 5

The assignment in parentheses is done first and its value is 4; 1 is added to it. That an assignment produces a value also makes chained assignments such as `x = y = 7;` possible. Even so, burying an assignment in the middle of an expression makes the code harder to read and is not preferred. Having a single effect on each line of a program makes both tracing and debugging easier.

> **Common mistake**
>
> Trying to capture in a variable the value that `println` prints: `int s = System.out.println(5);`. `println` writes to the screen but returns no value; the compiler reports `incompatible types: void cannot be converted to int`. If the result of a computation is both to be used and to be printed, assign it to a variable first and then print it.

2.1.8.3 From Fahrenheit to Celsius

The order of writing a program is fixed: first the algorithm, then the code. The algorithm breaks the problem into precise steps; the code writes those steps in the rules of the language. Let us build, in that order, a program that converts a temperature given in Fahrenheit to Celsius. The conversion formula is $C = (F - 32) \cdot 5 / 9$.

The algorithm has three steps:

```
1. Read the Fahrenheit value f.
2. Compute c = (f − 32) · 5 / 9.
3. Print c.
```

Each step has a counterpart in Java. `Scanner` is used for reading, an assignment statement for the computation, and `println` for printing:

```
import java.util.Scanner;

public class Fahrenheit {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Fahrenheit: ");
        double f = input.nextDouble();
        double c = (f - 32) * 5 / 9;
        System.out.println("Celsius: " + c);
    }
}
```

When the user enters 100, the screen shows:

```
Fahrenheit: 100
Celsius: 37.77777777777778
```

The code runs line by line as follows. The line `import java.util.Scanner;` makes the `Scanner` class available. `new Scanner(System.in)` creates an object that reads from the keyboard. Unlike `println`, `print` does not move to the next line, so the 100 the user types appears on the same line. `input.nextDouble()` reads a number and assigns it to `f`. In the computation line, first `f - 32` = 68.0 is found; because `f` is `double`, this result is `double` as well. Then 68.0 · 5 = 340.0 and 340.0 / 9 = 37.77777777777778 are computed.

The way the formula is written matters here too. If the same formula is written as `5 / 9 * (f - 32)`, the result is 0.0. The precedence rules have not changed: `5 / 9` is on the left and is done first. Since both operands are `int`, integer division gives 0, and 0 · 68.0 = 0.0. The compiler does not see this as an error; the program prints 0 °C for 100 °F. There are two ways to fix it: make the division floating-point (`5.0 / 9 * (f - 32)`) or move the multiplication first (`(f - 32) * 5 / 9`). The second works correctly as long as `f` is `double`; because `f - 32` produces a `double`, the division that follows is real division too.

Form	`f` is `double`	`f` is `int`
`(f - 32) * 5 / 9`	`37.77777777777778`	`37` (340 / 9, fraction discarded)
`5 / 9 * (f - 32)`	`0.0`	`0`
`5.0 / 9 * (f - 32)`	`37.77777777777778`	`37.77777777777778`

Table 2.1.14: Different ways of writing the Fahrenheit-to-Celsius formula, for F = 100.

The last column of the table shows how the type changes the result. Even if `f` is `int`, the `5.0` brings a `double` into the expression, so the result comes out right. The importance of the order of writing shows even in this small example: the algorithm is correct, and a small difference in notation during translation to Java spoils the result. The way to find such errors is to try an input whose result is known. It is known that 212 °F is 100 °C and 32 °F is 0 °C; if the program converts these two values correctly, the formula is most likely written correctly.

2.1.9 Popular programming languages

Programming languages offer different advantages for different tasks, and programmers choose the language that best fits the needs of a project. How much each language is used is tracked every year by surveys and indices. Stack Overflow's developer survey, the TIOBE index, and the IEEE Spectrum ranking are well known. Each uses a different measure: what survey respondents report, search-engine results, or job postings. The rankings therefore differ a little from one another, but the same languages are always near the top.

Language	Area of strength
Python	data analysis, artificial intelligence, web development
Java	large-scale enterprise applications, Android
JavaScript	web browsers, web development
C#	Windows applications, game development
C++	games, systems programming, performance-oriented software
Rust	safe and fast systems programming
Swift	application development on Apple platforms
Kotlin	Android application development (runs on the JVM)
Go (Golang)	simple and effective; network and cloud applications

Table 2.1.15: Some of the languages that stood out in 2023 and the areas in which they stand out.

The table can be read together with the classifications in the earlier sections. C, C++, Rust, and Go compile directly to machine code and stand out where performance matters. Java, Kotlin, and C# compile to an intermediate code and run on a virtual machine, balancing portability against speed. Python and JavaScript are usually interpreted and are preferred where fast development is needed. That Kotlin runs on the same JVM as Java lets the two languages be used together in one project.

From an engineering point of view there is no single “best language.” C may suit the software of a microcontroller, MATLAB or Python a signal-processing prototype, and Java a desktop measurement application. The concepts learned in one language (types, expressions, precedence, assignment, and statement order) carry over largely to the others. Learning a new language is often a matter of learning how these concepts are written in it and its small differences in rules.

2.1.10 Common mistakes

The mistakes of people new to programming come largely from the same few sources. The eight mistakes below summarize rules treated in detail in these notes. In each case the reaction of the compiler or the JVM was obtained with JDK 21.

Mistake	Example	Symptom	Fix
Forgetting a semicolon	`System.out.println("Hello")`	`';' expected`; `^` points to the end of the line	`;` at the end of the statement
Confusing `=` with `==`	`if (x = 100)`	`int cannot be converted to boolean`	`if (x == 100)`
Integer division	the average of 7 and 8 with `(a + b) / 2`	7 (expected 7.5); the compiler does not warn	`(a + b) / 2.0`
`**` or `^` for powers	`2 ** 3`; `2 ^ 3`	`illegal start of expression`; `^` compiles but is XOR (1)	`Math.pow(2, 3)`

Mistake	Example	Symptom	Fix
Order of string and number	`"Total: " + 2 + 3`	`Total: 23`	`"Total: " + (2 + 3)`
Uppercase versus lowercase	declaring `Count` and using `count`	`cannot find symbol`	write the name as it was declared
Using a variable before assigning it	`int s;` `System.out.println(s);`	`variable s might not have been initialized`	an initial value in the declaration: `int s = 0;`
Decimal separator	entering `7.5` into `nextDouble()` in the Turkish locale	`InputMismatchException`	enter `7,5` or use `useLocale(Locale.US)`

Table 2.1.16: Common mistakes, their symptoms, and their fixes.

The mistakes in the table fall into three groups. The semicolon, uppercase/lowercase, initial-value, and `=` / `==` mistakes are compile-time errors; the compiler catches them and its message points to the place. The decimal-separator mistake is a run-time error; the program compiles but crashes on a particular input. Integer division, powers with `^`, and concatenation order are logic errors. These last are the most dangerous, because the program produces a wrong result without any sign. The only way to find them is to try inputs whose results are known.

A general habit helps with compile-time errors: read the message carefully and start fixing from the first error. As in the `int = x 5;` example, a single mistake can lead to a series of messages. Once the first error is fixed, the others often disappear on their own. If nothing looks wrong on the line the message points to, look at the previous line too; an unclosed parenthesis or quotation mark in particular can carry the error several lines further on.

2.1.11 Summary

- A programming language is an artificial language designed to express computations that a machine can carry out. A program is both an instruction to the machine and a means of communication between programmers.
- Languages are classified by four criteria: level of abstraction (low or high level), application area (general-purpose or domain-specific), execution model (compiled or interpreted), and paradigm.
- In Java, `javac` compiles source code to bytecode (`.class`); the JVM interprets the bytecode and translates frequently executed parts to machine code with the JIT compiler. The same `.class` file runs on every platform that has a JVM.
- The four basic paradigms are imperative, functional, logic, and object-oriented. Java is fundamentally object-oriented and imperative; since Java 8 it also includes functional elements.
- Syntax is the formatting rules. Static semantics are the meaning rules for syntactically correct code that are checked at compile time (type compatibility). Semantics is what the code does; it is single and exact, but may not be what the programmer expects.

- There are four kinds of errors: syntax and static-semantics errors are found at compile time, run-time errors surface while the program runs as exceptions, and logic errors are found only by testing.
- Expressions are evaluated with parentheses first, then `* / %`, then `+ -`, left to right on the same level. The quotient of two integers is an integer and is truncated toward zero. Java has no exponentiation operator; use `Math.pow`.
- Assignment has two steps: the right-hand side is computed with the current values, and then written into the variable on the left. `=` is assignment and `==` is comparison.
- Java is statically typed: types are checked at compile time and do not change. Even `var` fixes the type at compile time. An operation on an `int` and a `double` is a `double` operation. If one operand is a string, `+` is concatenation and goes from left to right.
- Strings are compared with `equals`. Different types are first converted to the same type: `Integer.parseInt`, `String.valueOf`.
- A program consists of statements that run in order. An expression produces a value and a statement produces an effect. The order of statements determines the result; a swap is done with a third, temporary variable.
- A program is first designed as an algorithm and then translated into code. In the translation, details of type and precedence can change the result.

2.1.12 Key terms

programming language source code assembly language compiler interpreter bytecode
Java Virtual Machine (JVM) just-in-time (JIT) compiler programming paradigm
object-oriented programming field constructor reference syntax static semantics semantics
run-time error logic error exception expression operand operator precedence integer division
variable assignment identifier static / dynamic typing widening conversion type conversion
string string concatenation statement

2.1.13 Test Yourself

1. What are the values and types of the expressions `17 / 5 * 5 + 17 % 5` and `17 / 5.0`?

   Answer

   The first: `17 / 5` = 3 (integer division), 3 · 5 = 15, `17 % 5` = 2, total **17**, of type `int`. This result is no coincidence: `(a / b) * b + a % b` always equals `a`. The second: because 5.0 is `double`, real division is performed; **3.4**, of type `double`.

2. What does `System.out.println(2 * 3 + "x" + 2 * 3);` print?

> **Answer**
>
> `6x6`. The `*` operator has higher precedence than `+`; both multiplications are done before the concatenations, and the expression becomes `6 + "x" + 6`. Left to right: `6 + "x"` = `"6x"`, then `"6x" + 6` = `"6x6"`. The multiplications work correctly even without parentheses; the problem arises only when addition with `+` is what is wanted.

3. Suppose each line is in a separate program. What is the kind of error (syntax, static semantics, run-time, logic)? (a) `boolean b = 1;` (b) `int n = 5; int y = 10 / (n - 5);` (c) `double perimeter = 2 * pi + r;` for the perimeter of a circle

> **Answer**
>
> (a) Static semantics: `incompatible types: int cannot be converted to boolean`. In Java a `boolean` is not a number. (b) Run-time: the code compiles, but because `n - 5` = 0, `ArithmeticException: / by zero` is thrown. (c) Logic: the code compiles and runs, but the formula is wrong; the correct one is `2 * pi * r`.

4. After these lines run, what are `x` and `y`? `int x = 6; int y = x / 4; x = x * y + x % 4; y = x - y;`

> **Answer**
>
> `y` = 6 / 4 = 1 (integer division). The right-hand side of the third line is computed with the old `x` (6): 6 · 1 + 6 % 4 = 6 + 2 = 8; `x` = 8. The last line: 8 − 1 = 7. The result is `x` **= 8,** `y` **= 7**.

5. With `a` = 1, `b` = 2, `c` = 3, the values are to be rotated to the right: `a` ← `c`, `b` ← `a`, `c` ← `b`. In what order is this written with a temporary variable? What is the result?

> **Answer**
>
> First save the value that is to be overwritten first. The order `int t = c; c = b; b = a; a = t;` is correct: `t` = 3, `c` = 2, `b` = 1, `a` = 3. The result is **3 1 2**. Each assignment overwrites a variable whose value has already been copied elsewhere; no value is lost.

6. What does `IntStream.range(2, 6).sum()` give? How is the imperative equivalent of the same

computation written?

Answer

`range` excludes the upper bound: the numbers 2, 3, 4, 5 are added and **14** is found. The imperative equivalent: `int t = 0; for (int i = 2; i < 6; i++) t = t + i;`. The loop condition uses `<`, not `<=`; with `rangeClosed(2, 6)` the 6 would be included and the result would be 20.

7. With `int c = 37;`, what do the expressions `c * 9 / 5 + 32`, `9 / 5 * c + 32`, and `c * 9 / 5.0 + 32` give?

Answer

The first: 37 · 9 = 333, 333 / 5 = 66 (fraction discarded), 66 + 32 = **98**. The second: 9 / 5 = 1 (integer division), 1 · 37 + 32 = **69**; a large error. The third: 333 / 5.0 = 66.6, 66.6 + 32 = **98.6** (`double`), which is the correct value.

8. Let `x` be an `int` variable. Which of the following compile as a statement on their own? `x++;`, `x + 1;`, `Math.pow(x, 2);`, `x == 3;`

Answer

`x++;` compiles: it increments `x`, so it produces an effect. `Math.pow(x, 2);` compiles too: method calls can be statements, and the returned value is discarded. `x + 1;` and `x == 3;` do not compile, and both give the message `not a statement`; they only produce a value and have no effect.

2.1.14 Further reading

- Oracle, *The Java Tutorials*, "Language Basics": explains variables, operators, the precedence table, expressions, and statements with short examples. It is the official, free starting point for Java.
- A. B. Downey and C. Mayfield, *Think Java* (2nd ed., 2019), Chapters 1–2: covers programming languages, compilation and interpretation, kinds of errors, and reading error messages. The presentation is aimed at the beginning reader.
- J. V. Guttag, *Introduction to Computation and Programming Using Python* (3rd ed., 2021), Chapter 2: the source of the distinction among syntax, static semantics, and semantics. Because it gives the distinction with Python examples, it also lends itself to comparison with dynamic typing.
- J. Gosling et al., *The Java Language Specification, Java SE 21 Edition*, Chapter 15: defines the order of evaluation of expressions, integer division, and string concatenation with exact rules. The last word on the value of an expression is in this document.

2.1.15 References

1. P. Deitel and H. Deitel, *Java: How to Program, Early Objects*, 11th ed., Pearson, 2017 (Chapters 1–2).
2. A. B. Downey and C. Mayfield, *Think Java: How to Think Like a Computer Scientist*, 2nd ed., O'Reilly, 2019 (Chapters 1–2).
3. J. Gosling et al., *The Java Language Specification, Java SE 21 Edition*, Oracle, 2023 (Chapter 3: Lexical Structure; Chapter 15: Expressions).
4. J. Guttag, *Introduction to Computation and Programming Using Python*, 3rd ed., MIT Press, 2021 (Chapter 2: syntax, static semantics, and semantics).
5. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson, 2022 (Chapters 2–3).
6. Oracle, *The Java Tutorials*: "Language Basics" (variables, operators, expressions, statements, and blocks).
7. K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly, 2022.

PART I · CHAPTER 2.2

Syntax Rules

A Java program is handed to the compiler as text. The compiler reads this text character by character and checks whether it follows the rules of the language. If a single semicolon is missing, or a letter is lowercase where it should be uppercase, the program will not run. These notes describe the writing rules of Java and the machinery behind them. The first section follows a `.java` file as it becomes bytecode through `javac` and then a running program through the `java` command. It explains how the JVM works on its stack and why the same `.class` file runs on every operating system. Next come variables, the eight primitive types, and the limits of those types. Identifiers, keywords, and naming conventions determine how names are chosen. The section on assignment, literals, and type conversion shows how numbers change when they move from one type to another. The last sections cover operators, operator precedence, short-circuit evaluation, the conditional operator, and the syntax of control statements. In every section the code from the slides is explained line by line, and the compiler's real error messages are shown.

LEARNING OUTCOMES

Compiling and running

Describe how a `.java` file becomes bytecode with `javac` and a running program with `java`; relate the levels of abstraction; trace bytecode on the JVM stack; explain how the JVM provides platform independence.

Variables and types

Know the eight primitive types with their sizes and ranges; declare variables and assign values; notice overflow and floating-point precision problems.

Rules and naming

Tell a valid identifier from an invalid one, recognize keywords, and follow the camelCase, PascalCase, and UPPER_CASE conventions.

Expressions and flow
Evaluate expressions by operator precedence, perform type conversion; choose a value by a condition with `?:`; recognize the syntax of the `if`, `switch`, `for`, `while`, `do-while`, `break`, and `continue` statements.

2.2.1 Compiling and running

A program written in Java goes through several stages before a result appears on the screen. The programmer writes the code in a text file. The compiler checks this text and translates it into an intermediate code. The Java Virtual Machine takes the intermediate code and executes it on the computer's processor. This section takes the stages in order. Reading error messages correctly, and knowing what happens behind the "Run" button, become possible only once this chain is familiar.

2.2.1.1 The source file

Definition 2.2.1

Source code is the human-readable program text that a programmer writes according to the rules of a programming language. Java source code lives in **plain text** files, and these files have the extension `.java`.

Plain text is a file made up of characters only. It carries no information about typeface, color, bold type, or page layout. A document saved by a word processor such as Word stores this formatting information next to the text, and the compiler does not understand that extra information. Java code is therefore written in a text editor such as Notepad, Notepad++, or VS Code, or in a development environment.

Plain text is convenient for three reasons:

- **It is readable.** A person who opens the file sees the code directly. The code is easy to understand, can be sent by e-mail, and can be kept in a version control system.
- **It is portable.** The same `.java` file opens on every operating system and in every development environment.
- **It is easy to edit.** Any text editor is enough to fix a bug or update the code.

Java also imposes a rule on file names. If a file contains a `public` class, the file's name must be exactly the same as the class name. Code that begins with `public class HelloWorld` is saved in a file named `HelloWorld.java`. Uppercase and lowercase matter in this match too. If the same code is saved as `helloworld.java`, the compiler gives this message:

```
helloworld.java:1: error: class HelloWorld is public, should be declared in a file named
HelloWorld.java
public class HelloWorld {
       ^
1 error
```

Although the Windows file system does not distinguish uppercase from lowercase, `javac` does. The message says exactly what is wrong: the class `HelloWorld` is public and must be declared in a file named `HelloWorld.java`.

2.2.1.2 Programming tools

Several tools are used together to write a Java program. Table 2.2.1 summarizes these tools and their roles.

Tool	What is it for?	Examples
Text editor	Writing source code as plain text	Notepad++, VS Code
JDK (Java Development Kit)	The compiler, the launcher, and other development tools	`javac`, `java`, `javap`, `jshell`
IDE (integrated development environment)	Editing, compiling, running, and debugging in one window; errors are underlined as you type	Eclipse, IntelliJ IDEA, NetBeans, VS Code with Java extensions
JShell (Java 9 and later)	Trying out one-line Java expressions without writing a class	`jshell> 7 / 2` → `3`

Table 2.2.1: Tools used to develop programs in Java.

The **JDK** (Java Development Kit) is the package that contains everything needed to develop programs in Java. It includes the compiler `javac`, the `java` command that starts a program, `javap`, which shows the inside of a `.class` file, and the interactive experimentation environment `jshell`. An **IDE** (integrated development environment) combines these tools with an editor. When you press the "Run" button in an IDE, `javac` and `java` still run in the background.

JShell is very handy for small experiments. Someone who wonders why `7 / 2` gives 3 and not 3.5 can type the expression into JShell without writing a class and a `main` method, and see the result at once. It is also worth using `javac` and `java` from the command line at least once at the start. The steps an IDE hides then become visible, and when an error appears you can tell at which stage it appeared.

2.2.1.3 The first program: HelloWorld

The program below is one of the shortest complete programs that can be written in Java. It prints a single line to the screen.

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Short as it is, every part of the program follows a rule:

- `public class HelloWorld` defines a class. Every Java program contains at least one class. The body of the class is the part between the `{` on this line and the `}` on the last line. Because the class is public, the file must be named `HelloWorld.java`.
- `public static void main(String[] args)` is the **entry point** of the program. When the JVM starts the program, it begins running from this method. The header must be written exactly this way. If `Main` or `main(String args)` is written, the program compiles, but the JVM cannot find the entry point and refuses to run it.
- `System.out.println(...)` writes the value in parentheses to the console and moves to the next line. Text is written in double quotes: `"Hello, World!"`.
- The **semicolon** at the end of the line shows that the statement is over. Every statement ends with `;`. Blocks closed by braces need no `;` after them, which is why there is no semicolon after the closing brace of the class or of the method.

The body of the program consists of two nested blocks. The outer block is the body of the class and the inner block is the body of the `main` method. Indentation, that is, the spaces at the beginning of the lines, makes this nesting visible to the eye. The compiler does not look at indentation; it only counts braces.

2.2.1.4 Basic syntax rules

> **Definition 2.2.2**
>
> **Syntax** is the set of writing rules of a language: it determines which words, symbols, and orderings are valid. Java code that breaks the syntax rules does not compile.

In English, the phrase “Book the read I” has an unusual order but can be understood. The compiler is not so tolerant. It does not correct a line that breaks the rules by guessing; it reports an error and stops. The basic writing rules of Java are:

- Every statement ends with `;`.
- **Blocks** are opened with `{` and closed with `}`. Every block that is opened must be closed.
- Java is **case-sensitive**: `num` and `Num` are two different names; `system` cannot be written in place of `System`.

- Spaces, tabs, and line breaks are unimportant to the compiler. A statement can be split over several lines. Indentation is used for readability.
- **Comments** are written in two forms: `//` runs to the end of the line, and `/* … */` can span several lines. The compiler ignores comments.

The program below shows all of these rules together.

```
/* Multi-line comment:
   the compiler ignores it. */
public class Rules {
    // Single-line comment
    public static void main(String[] args) {
        int num = 5;  // ends with ;
        int Num = 7;  // num ≠ Num
        System.out.println(num
                + Num);   // a line can be split
    }
}
```

The program gives this output:

```
12
```

The first two lines form a multi-line comment and add nothing to the compiled code. `num` and `Num` are two separate variables because of the difference in case: one holds 5 and the other 7. The `println` call is split over two lines. For the compiler this split is unimportant, because a statement ends only when a `;` is seen. The 12 in the output is the sum of the two variables. That two variables can sit side by side shows how important a single letter in a name can be. In real programs, such similar names cause confusion and should be avoided; the aim here is only to show the rule.

Common mistake

Mixing up uppercase and lowercase. The three lines below are typical mistakes caused by habits of writing everything in lowercase. The compiler gives a separate message for each:

```
Case.java:3: error: package system does not exist
        system.out.println("a");
              ^
Case.java:4: error: cannot find symbol
        string s = "b";
        ^
  symbol:   class string
  location: class Case
Case.java:5: error: cannot find symbol
        Int n = 3;
        ^
  symbol:   class Int
  location: class Case
3 errors
```

When `system.out` is written, the compiler takes `system` to be a package name and cannot find that package. `string` and `Int` look like class names that are not defined. The correct forms are `System`, `String`, and `int`. `String` starts with an uppercase letter because it is a class, and `int` with a lowercase letter because it is a keyword.

2.2.1.5 Levels of abstraction

A processor understands only commands encoded as binary numbers. A person thinks more like "add a and b and store the result in sum." On the way from person to processor there are three levels, and each level hides the details of the one below it.

- A **high-level language** is close to the problem domain. Java, C, and Python are at this level. The code is short, readable, and understandable, and the programmer works more productively.
- **Assembly language** writes processor instructions with short names such as `MOV` and `ADD`. Each line often corresponds to a single machine instruction. It is a lower level of abstraction.
- The **hardware representation** consists only of bits. Instructions and data go directly to the hardware in this form.

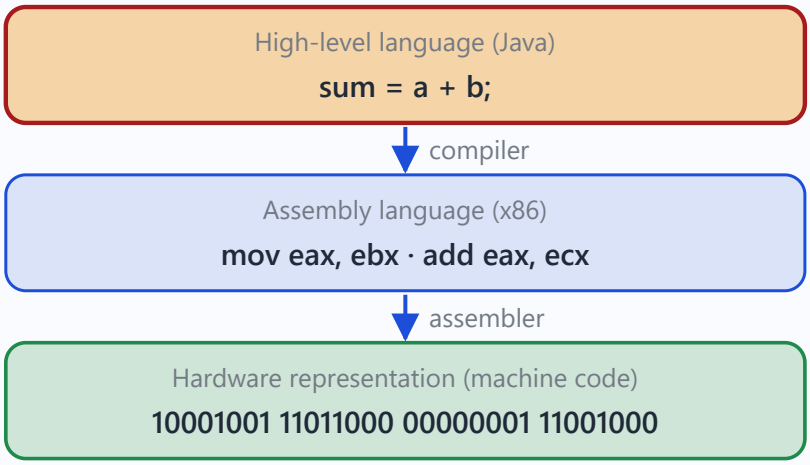


Figure 2.2.1: Three levels of the same addition. The single line in Java corresponds to two instructions in assembly language and to two machine instructions of two bytes each in hardware.

Figure 2.2.1 shows the counterparts of the line `sum = a + b` at the three levels. The example assumes that the variables `a`, `b`, and `sum` live in the registers `ebx`, `ecx`, and `eax` respectively. In assembly language, the value of `a` is first copied into `eax` (`mov eax, ebx`), and then `b` is added on top of it (`add eax, ecx`). The machine code is the real encoding for x86 processors: the first instruction is the hexadecimal bytes `89 D8` and the second is `01 C8`. As abstraction increases, the code gets shorter and more readable. The translation to the lower level is done by a compiler or an assembler.

Java adds an intermediate layer to this picture. The Java compiler does not translate source code directly to x86 machine code. It first translates it to a code called bytecode, which belongs to no real processor. This intermediate code allows the same compiled program to run on different processors.

2.2.1.6 Compilation

Definition 2.2.3

Compilation is the translation of source code, as a whole, into another language. The program that does this is called a **compiler**. The Java compiler `javac` translates source code not into the machine language of a processor but into **bytecode**, which the Java Virtual Machine understands, and writes the result to a file with the extension `.class`.

A computer cannot run `.java` text directly. The text has to be compiled first. The path from source code to the output on the screen is completed in four steps:

1. The programmer writes the source code as plain text and saves it as `HelloWorld.java`.
2. The command `javac HelloWorld.java` runs the compiler. The compiler checks the syntax and the types. If there are no errors, the file `HelloWorld.class` is created in the same folder.

3. The command `java HelloWorld` runs the **launcher**. The launcher starts the JVM and loads the class. The JVM interprets the bytecode and translates the frequently executed parts into machine code with the JIT compiler.
4. The operating system and the processor carry out the actual instructions; `Hello, World!` appears on the screen.

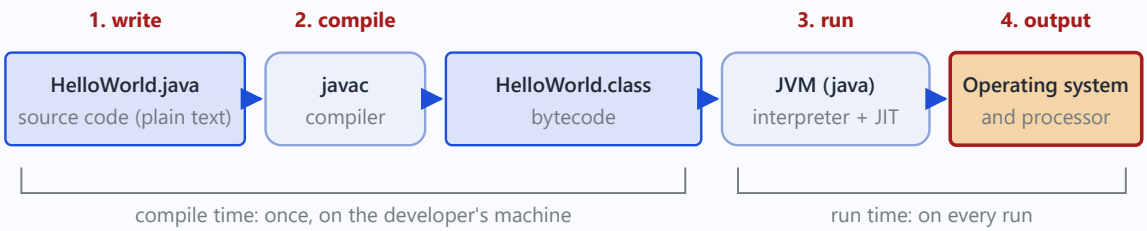


Figure 2.2.2: The compile-and-run pipeline. The `.java` file becomes a `.class` file with `javac`, and the `.class` file becomes a running program with the JVM.

In Figure 2.2.2 there are two separate commands and two separate periods of time. **Compile time** is the time during which `javac` runs; during it the program does not run yet, it is only checked and translated. **Run time** begins with the `java` command. It matters at which time an error appears. A compile-time error means the program never runs at all. A run-time error appears while the program is running; division by zero is an example.

The greatest benefit of the compilation stage is that errors are caught before the program runs. The compiler finds a missing semicolon, an unclosed parenthesis, a misspelled name, and incompatible types. If there is an error, no `.class` file is produced. The `file name:line` information in the message shows where the error is. If the semicolon on line 3 of the `HelloWorld` program is forgotten, the line becomes:

```
System.out.println("Hello, World!")
```

For this line the compiler gives this message:

```
HelloWorld.java:3: error: ';' expected
        System.out.println("Hello, World!")
                                           ^
1 error
```

The first line of the message gives the file (`HelloWorld.java`), the line number (3), and the kind of error (`';' expected`, that is, a semicolon was expected). The second line repeats the faulty code. The `^` mark on the third line shows where the compiler expected the semicolon: immediately to the right of the closing parenthesis. The last line reports the total number of errors. The compilation stage also

contributes to the speed of the program. The compiler does jobs such as type checking once, so they need not be repeated while the program runs. The JVM's optimizations at run time come on top of this.

> **Common mistake**
>
> Not looking before the line that the message points to. When a semicolon is missing, the compiler usually notices it only when it reads the next token, so the message sometimes points one line down. If a puzzling `';' expected` message appears on a line, first look at the end of the previous line.

2.2.1.7 Compiling and running from the command line

There are three basic steps in working with Java. In the **compile** step the `.java` file is turned into a `.class` file: `javac HelloWorld.java`. In the **run** step the `java` command is given the class name without an extension: `java HelloWorld`. The third step is the **output**: the program prints `Hello, World!` on the screen. A command-line session looks like this:

```
> javac HelloWorld.java
> dir /b
HelloWorld.class
HelloWorld.java
> java HelloWorld
Hello, World!
```

If `javac` printed nothing, the compilation succeeded. The `dir /b` command that follows (`ls` on Linux and macOS) lists the files in the folder. The list shows the newly created `HelloWorld.class` next to the source file. The last command runs the program.



Figure 2.2.3: Two commands, two files: `javac` turns the source file into a `.class` file; `java` starts the JVM and the program produces output.

> **Common mistake**
>
> Writing the extension when running: `java HelloWorld.class`. The JVM expects a **class name**, not a file name. This command looks for a class named `HelloWorld.class` and stops with the message "Could not find or load main class." The correct form is without the

extension: `java HelloWorld`.

Java 11 added a shortcut for single-file programs. The command `java HelloWorld.java` compiles the file in memory and runs it at once. No `.class` file is written to disk this way. The shortcut is practical for small experiments, but programs made of several files use the two-step route.

2.2.1.8 Bytecode

Definition 2.2.4

Bytecode consists of the instructions, independent of processor and operating system, that `javac` produces and the Java Virtual Machine executes. `.class` files do not contain machine code that a processor can understand directly; they contain bytecode. Bytecode is the "machine language" of the JVM.

The name comes from the structure of the instructions: the **opcode** of each instruction is a single byte. Any extra information the instruction needs, such as a constant or the number of an entry in a table, sits in the bytes after the opcode. A one-byte opcode means at most 256 different instructions, and that is enough for the JVM.

Bytecode is not tied to a processor or an operating system. The same `.class` file runs on Windows, on Linux, and on macOS. An analogy describes the relationship well: bytecode is like a musical score that reads the same in every country, and the JVM on each computer is the musician who plays that score on its own instrument. The score does not change; the instrument and the musician do.

A `.class` file is a binary file and looks like meaningless characters in a text editor. The command `javap -c ClassName` in the JDK lists the bytecode instructions in this file in readable form. Consider this two-line code: `int x = 5; System.out.println(x);`. When the code is placed inside a `main` method and compiled with JDK 21, `javap -c` gives this listing:

```
0: iconst_5
1: istore_1
2: getstatic     #7    // System.out
5: iload_1
6: invokevirtual #13   // println(int)
9: return
```

The number at the beginning of each line is the instruction's byte position within the method. The positions go 0, 1, 2, 5, 6, 9. `iconst_5` and `istore_1` are one-byte instructions. `getstatic` and `invokevirtual` carry a two-byte table number after the opcode, so each takes three bytes.

The JVM is a **stack machine**. Real processors keep values in registers. The JVM's instructions push values onto an **operand stack** and take them from it. A stack is like plates piled on top of one another: the one put on last is taken first. The instructions do the following in order:

1. `iconst_5`: the constant 5 is pushed onto the operand stack. Its counterpart in the source is the right-hand side of `int x = 5;`.
2. `istore_1`: the value on top of the stack is taken and written to local variable number 1, that is, `x`. The stack is empty again.
3. `getstatic #7`: a reference to the `System.out` object is pushed onto the stack. `#7` is entry number 7 in the class file's constant pool.
4. `iload_1`: the value of `x` (5) is copied onto the stack. Its counterpart in the source is the line `System.out.println(x);`.
5. `invokevirtual #13`: the method `println(int)` is called. The method takes two values from the stack: the object it is to work on (`System.out`) and the number to print (5). 5 is written to the screen.
6. `return`: `main` ends, and the program terminates.

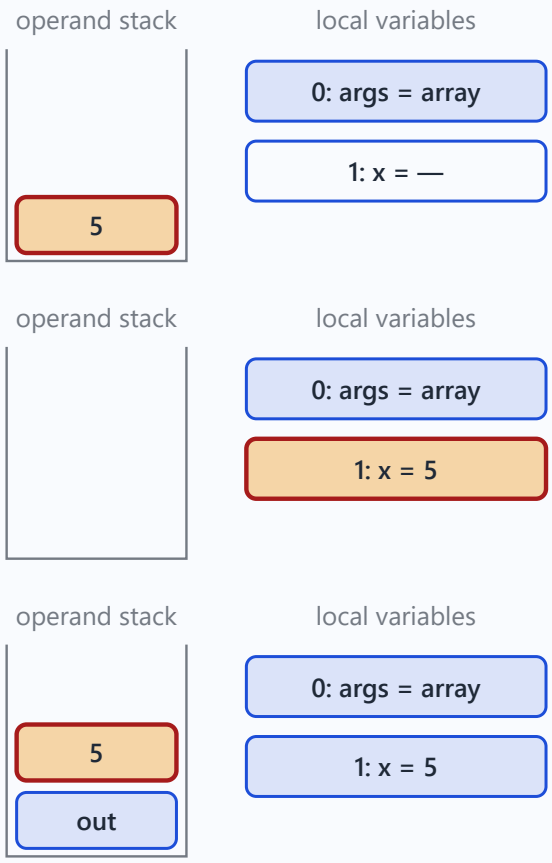


Figure 2.2.4: The operand stack and the local variables while bytecode runs. (a) After `iconst_5` the stack holds 5 and `x` is still empty. (b) After `istore_1` the stack is empty and `x` = 5. (c) After `iload_1` the stack holds the `System.out` reference and 5; `println` will take these two.

Figure 2.2.4 shows three moments. The difference between (b) and (c) matters: `iload_1` **copies** the value from the local variable onto the stack. `x` stays where it is and its value does not change. Reading a variable does not consume it; this holds at the bytecode level too.

2.2.1.9 An addition on the JVM stack

The stack-machine idea is clearer in an addition. Consider these three lines:

```
int a = 5;
int b = 7;
int sum = a + b;
System.out.println(sum);
```

The compiled form of these lines is twelve instructions. The comments give the source-code counterpart of each instruction:

```
iconst_5        // push 5
istore_1        // a = 5
bipush 7        // push 7
istore_2        // b = 7
iload_1         // load a
iload_2         // load b
iadd            // add
istore_3        // sum = 12
getstatic       // System.out
iload_3         // load sum
invokevirtual   // println(int)
return
```

The JVM uses no register to add. It pushes the two numbers onto the stack, the `iadd` instruction pops the top two values and adds them, and it pushes the result back onto the stack. Table 2.2.2 gives the operand stack and the local variable table after each instruction. Slot 0 of the local variable table holds `main`'s parameter `args`; the table shows it as "…". `a`, `b`, and `sum` take slots 1, 2, and 3 respectively.

Instruction	Operand stack	1: `a`	2: `b`	3: `sum`	Explanation
`iconst_5`	5				the constant 5 is pushed
`istore_1`	empty	5			`a = 5`
`bipush 7`	7	5			the constant 7, which fits in one byte, is pushed
`istore_2`	empty	5	7		`b = 7`
`iload_1`	5	5	7		`a` is copied
`iload_2`	5, 7	5	7		`b` is copied
`iadd`	**12**	5	7		two values come off, the sum goes on

Instruction	Operand stack	1: `a`	2: `b`	3: `sum`	Explanation
`istore_3`	empty	5	7	**12**	`sum = 12`
`getstatic`	out	5	7	12	the `System.out` reference
`iload_3`	out, 12	5	7	12	the argument of `println`
`invokevirtual`	empty	5	7	12	12 is written to the screen
`return`	empty	5	7	12	`main` ends

Table 2.2.2: The state of the JVM after each instruction in the example `int sum = a + b;`. The top of the stack is on the right.



Figure 2.2.5: Three moments of the addition. (a) After `iload_2` the stack holds 5 and 7. (b) `iadd` pops both and pushes 12. (c) `istore_3` writes 12 into the slot of `sum` and the stack becomes empty.

Three observations follow from the table. First, `iconst_5` is used for 5, but `bipush 7` for 7. In the JVM each small constant from −1 to 5 has its own one-byte instruction. Since 7 is not in this range, `bipush`, meaning "push a byte," is written, followed by a byte carrying the value 7. Second, the stack is emptied at the end of every statement. The stack stays full only for the short time during which an expression is being computed. Third, the `iload` instructions do not consume the variables; `a` and `b` remain 5 and 7 after the addition.

The letter `i` in the instruction names indicates the type `int`. There are separate instructions such as `ladd` for `long` and `dadd` for `double`. The JVM knows which type it is working with in every operation. This is one reason types are checked at compile time: the compiler decides which instruction to generate by looking at the type. The same bytecode runs in this order on the JVM of every operating system.

2.2.1.10 The Java Virtual Machine and the JIT compiler

Definition 2.2.5

The **Java Virtual Machine** (JVM) is the software that runs Java applications. It interprets bytecode and executes it in a way suited to the processor it is on. It translates frequently executed pieces of bytecode into machine code at run time with the **JIT** (just-in-time) compiler.

The name "virtual machine" comes from the fact that the JVM behaves like a computer with its own instruction set. This computer does not exist as hardware; it runs as a program on top of the real computer. The internal structure of the JVM has four main parts:

- The **class loader** finds the needed `.class` files and brings them into memory.
- The **verifier** checks that the loaded bytecode follows the rules. For example, it makes sure that an instruction does not try to take a value that is not on the stack. A corrupted or malicious file therefore cannot crash the JVM.
- The **interpreter** reads the bytecode instructions one by one and does the work corresponding to each.
- The **JIT compiler** notices code fragments that run many times and translates them directly into machine code. In later runs this machine code is used instead of interpretation.

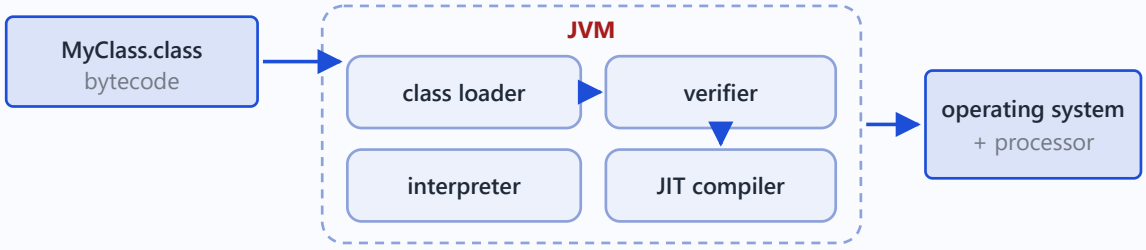


Figure 2.2.6: The internal structure of the JVM: class loader, verifier, interpreter, and JIT compiler. The work produced goes to the operating system.

Interpretation is flexible but slow: every instruction is decoded again each time it is run. Machine code is fast, but producing it takes time. The JIT strikes a balance between the two. Code that runs once is interpreted; a loop body that runs thousands of times is compiled. A Java program therefore

speeds up over time. This effect is noticeable in long-running server programs.

The `java` command that starts the program is called the **Java launcher tool** in the JDK documentation. The launcher starts the JVM, loads the class whose name is given, and calls the `main` method of that class. When `main` ends, the JVM shuts down too.

2.2.1.11 Platform independence and the Java platform

> **Definition 2.2.6**
>
> A **platform** is the hardware or software environment in which a program runs: the combination of a processor and an operating system. **Platform independence** means that the same compiled program runs, without being recompiled, on every platform that has a JVM.

Platform independence is one of the fundamental features of Java and is known by the slogan "Write Once, Run Anywhere." The program is compiled once, and the same bytecode runs on different operating systems. What makes this possible is that JVMs have been written separately for many operating systems and processors. A JVM written for Windows translates bytecode into x86 machine code. The JVM on a Mac with an ARM processor translates the same bytecode into ARM machine code. The difference is in the JVM, not in the bytecode.

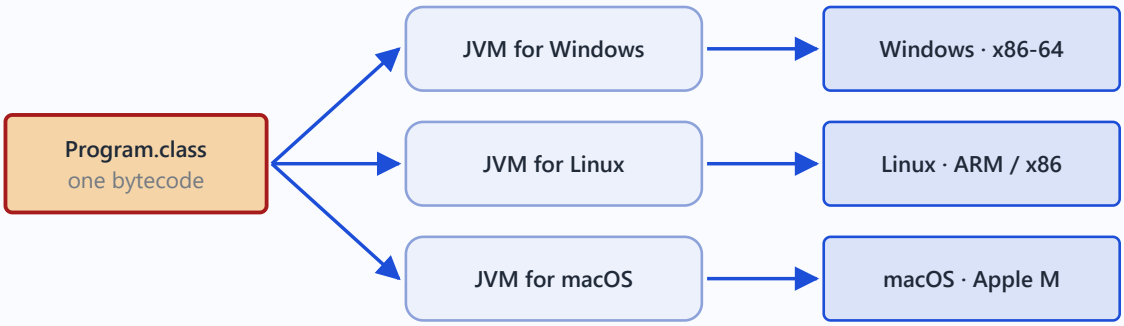


Figure 2.2.7: Platform independence. The same `.class` file runs on JVMs written for Windows, Linux, and macOS.

In languages such as C the situation is different. A C compiler produces machine code directly, and this code is tied to the platform it was compiled on. A C program compiled on Windows does not run on Linux; a separate build is needed for every platform. In Java this burden passes to those who write the JVMs. The programmer distributes a single `.class` file, and it is enough for the user's computer to have a JVM.

The Java platform has two main components:

- The **JVM** runs applications and provides portability across different hardware.

- The **API** (Application Programming Interface) consists of ready-made class libraries. It provides many functions out of the box, such as graphical interfaces, databases, network communication, and file operations, and speeds up development. `System`, `String`, `Math`, and `Scanner` are classes in these libraries.

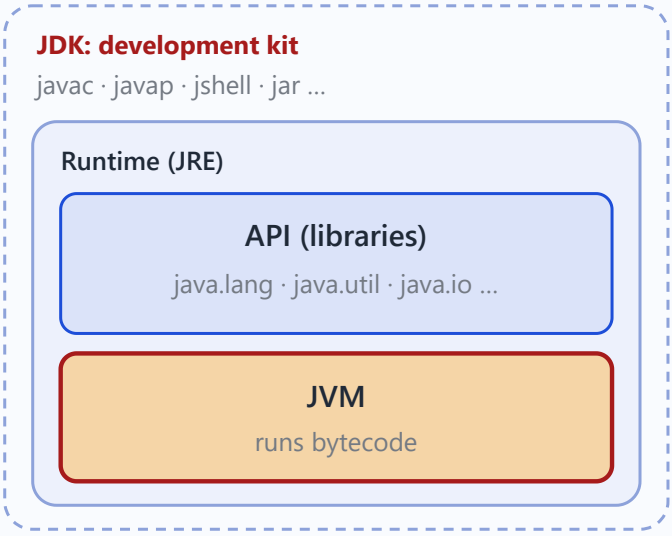


Figure 2.2.8: The layers of the Java platform. The JDK contains the runtime environment, and the runtime environment contains the API libraries and the JVM.

Figure 2.2.8 shows that the layers are nested. The runtime environment (JRE, Java Runtime Environment) contains what is needed to run a Java program, namely the JVM and the API libraries. The JDK adds the compiler and the other development tools on top of these. To develop programs, the JDK is installed. Since Java 11, Oracle has not distributed the JRE as a separate package; the JDK contains everything.

2.2.2 Variables and data types

A program takes in data, processes it, and produces a result. At every moment of this process the data has to be held somewhere. Variables do this job. Java wants to know in advance what kind of data each variable holds; this information is called the type. This section covers variables, Java's eight primitive types, and the limits of each type.

2.2.2.1 Basic programming elements

Every program, in whatever language it is written, is built from a few basic elements. **Variables** store data. Every variable has a **type**. Values are computed with **expressions**: `num > 0` and `sum + 1` are expressions. **Control flow** determines in what order and how the program runs. Control flow has two basic forms:

- **Branching** runs different parts of the program depending on a condition. `if-else` is an example.
- **Loops** repeat an operation. `for` and `while` are examples.

The following fragment brings together a variable, a condition, and output:

```
int num = 5;
if (num > 0) {
    System.out.println("Positive.");
} else {
    System.out.println("Zero or negative.");
}
```

```
Positive.
```

Because `num` is 5, the expression `num > 0` is true and the first block runs. The `else` block is skipped. If `num` were 0 or negative, the first block would be skipped and the second would run. Here the value of the expression `num > 0` is not a number but a piece of true-or-false information. Java uses a separate type for this information: `boolean`.

2.2.2.2 Variables and declarations

Definition 2.2.7

A **variable** is a location in memory that stores a value and has a name and a type. Introducing a variable before it is used is called a **declaration**. In a declaration the type is written first, then the name, and the declaration ends with `;`.

A variable can be pictured as a box with a name written on it. The label on the box carries two pieces of information: its name and what kind of thing may be put in it. Only an integer goes into a box labeled `int`, and only a floating-point number into a box labeled `double`. The value inside the box can change while the program runs; its name and type do not.



Figure 2.2.9: Variables are like labeled boxes: `id` (`int`) holds 1042, `interestRate` (`double`) holds 0.45, and `answer` (`char`) holds `'Y'`.

In Java every variable must be declared before it is used. Several variables of the same type can be declared on one line, separated by commas:

```
int id, checkCount, stockCount;
double amount, interestRate;
char answer;
```

The first line declares three `int` variables, the second two `double` variables, and the third one `char` variable. A declaration has three forms: `type name;`, `type name1, name2, name3;`, and `type name = initialValue;`. The last also gives the variable its first value together with the declaration.

Names should be chosen to be descriptive and meaningful. The name `x` says nothing about what the variable holds. `checkCount`, on the other hand, makes it clear at once that it holds the number of checks in an account. The person who rereads the code a few months later is often the person who wrote it. Meaningful names make that person's job easier.

2.2.2.3 Declaration, assignment, update

A variable's life proceeds through three events: it is declared, it receives a value, and that value is updated. The code below contains all three events:

```
int age;
age = 25;
double height = 1.78;
age = age + 1;
char letter = 'K';
System.out.println(age + " " + height + " " + letter);
```

The program prints `26 1.78 K`. Table 2.2.3 shows the boxes in memory after each line.

Line	Code	`age`	`height`	`letter`	Explanation
1	`int age;`	?			a box is set aside, empty
2	`age = 25;`	**25**			assignment: 25 is put in the box
3	`double height = 1.78;`	25	**1.78**		declaration and initial value on one line
4	`age = age + 1;`	**26**	1.78		first 25 + 1 = 26 is computed
5	`char letter = 'K';`	26	1.78	**'K'**	a character is written in single quotes
6	`println(…)`	26	1.78	'K'	output: `26 1.78 K`

Table 2.2.3: Trace of declaration, assignment, and update. "?" marks a box with no value assigned, and an empty cell marks a variable not yet declared.

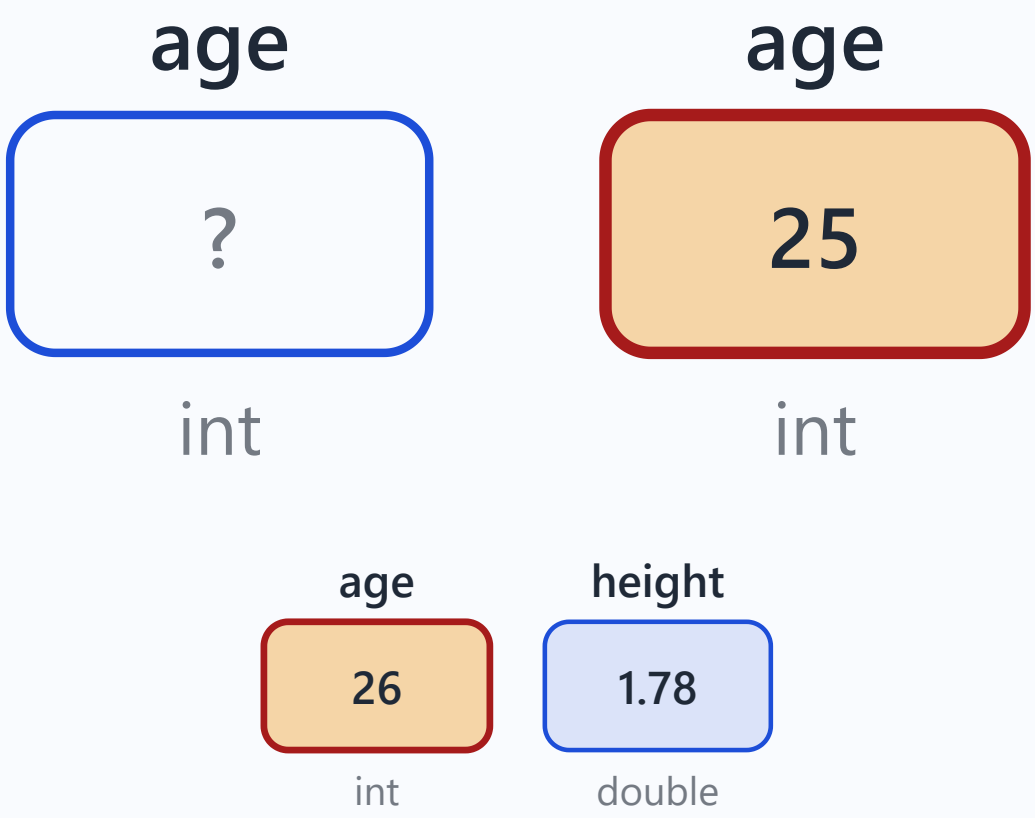


Figure 2.2.10: Three moments in the life of a variable. (a) `int age;` sets the box aside, empty. (b) `age = 25;` puts 25 in the box. (c) After `age = age + 1;` the box holds 26; the old value is gone.

After line 1 there is a box called `age` but no value in it. In this state `age` cannot be read; code that tries to read it does not compile. Line 4 is an update. The right-hand side is computed first: the 25 in the box is read and 1 is added. The result, 26, is written into the same box and the old value is lost. In the expression `age + " " + height + " " + letter` on line 6, the `+` sign does not add numbers. Because one of the operands is text, it appends the values to the text. This operation is called **string concatenation**.

2.2.2.4 The eight primitive types

Definition 2.2.8

A **data type** determines which values a variable can take, how much memory it occupies, and which operations can be performed on it. Java's eight basic types built into the language are called **primitive types**: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`.

The eight types fall into four groups:

- **Integer:** `byte`, `short`, `int`, `long`.
- **Floating-point:** `float`, `double`.
- **Character:** `char`.
- **Logical:** `boolean`.

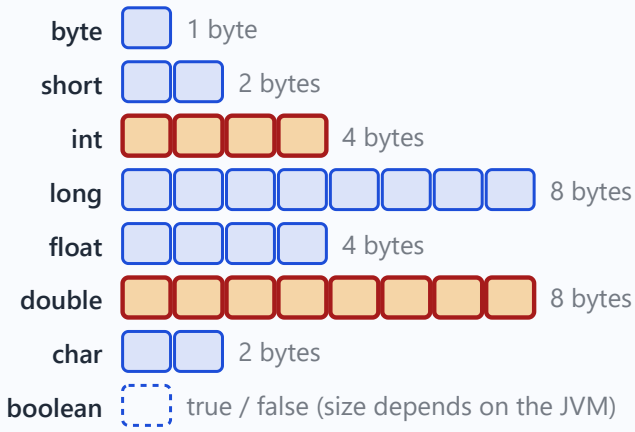


Figure 2.2.11: The memory occupied by the primitive types: `byte` 1, `short` 2, `int` 4, `long` 8, `float` 4, `double` 8, `char` 2 bytes. The size of `boolean` is determined by the JVM.

Each type has its own size in memory and a range of values that follows from it. As the number of bits grows, a type can hold larger numbers but takes more memory. In C the size of `int` can vary with the compiler and the processor. In Java, sizes are fixed by the definition of the language: `int` is 32 bits on every platform. This fixedness is part of platform independence too.

In everyday use, `int` is the default choice for integers and `double` for floating-point numbers. `byte` and `short` are mostly used in special cases, such as saving memory in large arrays or reading bytes from a file. `float` is rarely preferred outside graphics and some embedded systems.

2.2.2.5 Integer types

Type	Memory	Smallest value	Largest value	Example
`byte`	1 byte (8 bits)	−128	127	`byte b = 100;`
`short`	2 bytes (16 bits)	−32,768	32,767	`short s = 32000;`
`int`	4 bytes (32 bits)	−2,147,483,648	2,147,483,647	`int n = 1_000_000;`
`long`	8 bytes (64 bits)	−9,223,372,036,854,775,808	9,223,372,036,854,775,807	`long l = 3000000000L;`

Table 2.2.4: Integer types, their memory sizes, and their ranges.

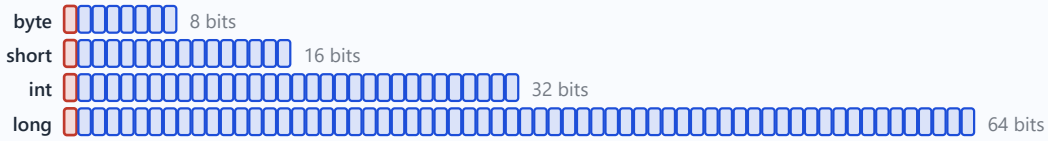


Figure 2.2.12: The bit widths of the integer types: `byte` 8, `short` 16, `int` 32, `long` 64 bits.

All four types are **signed**: they take both positive and negative values. The range of an n-bit type is from -2^{n-1} to $2^{n-1} - 1$. For `byte`, $n = 8$, and the range lies between $-2^7 = -128$ and $2^7 - 1 = 127$. The positive side is one short because zero also takes a place in the positive half: 8 bits give 256 different values; 128 of them are negative, one is zero, and 127 are positive.

There is no need to memorize the limits. Java provides constants for every type: `Byte.MIN_VALUE`, `Byte.MAX_VALUE`, `Integer.MAX_VALUE`, `Long.MIN_VALUE`, and so on. Still, it is useful to know that the upper limit of `int` is about 2.1 billion. The world's population, a month measured in milliseconds, or a country's budget exceeds this limit.

An integer written directly in code, such as `42`, is taken to be an `int`. To write a number beyond the `int` range as a `long`, add `L` to the end: `3000000000L`. A lowercase `l` is valid too, but it looks so much like the digit 1 that the uppercase letter is preferred. If the `L` is forgotten, the compiler gives the message `integer number too large`: before the number is assigned to the `long` variable, it is read as an `int` and does not fit.

2.2.2.6 Floating-point types

Numbers with fractional parts are stored in a computer in **floating-point** form. This form is the binary-base counterpart of scientific notation. A number consists of a sign, an exponent, and a fraction: in the number 6.02×10^{23}, 6.02 is the fraction and 23 is the exponent. The fraction holds the significant digits and the exponent determines where the point goes. The point "floats" according to the magnitude of the number; hence the name.

Type	Memory	Smallest positive	Largest (±)	Significant digits
`float`	4 bytes (32 bits)	$\approx 1.4 \times 10^{-45}$	$\approx 3.4028235 \times 10^{38}$	≈ 7
`double`	8 bytes (64 bits)	$\approx 4.9 \times 10^{-324}$	$\approx 1.7976931348623157 \times 10^{308}$	≈ 15–16

Table 2.2.5: Floating-point types. "Smallest positive" is the smallest positive value other than zero.



Figure 2.2.13: The bits of a floating-point number. `float`: 1 sign bit, 8 exponent bits, 23 fraction bits. `double`: 1 sign bit, 11 exponent bits, 52 fraction bits.

Figure 2.2.13 shows the difference between the two types. The fraction part of `double` is 52 bits, which corresponds to about 15–16 decimal digits. The 23-bit fraction of `float` holds about 7 digits. `double` is **double precision**: compared with `float` it has both a wider range and more digits. The negative limits in the table are the largest values with a minus sign.

A floating-point number written in code, such as `3.14`, is taken to be a `double`. For a `float`, add `f` at the end: `3.14f`. The difference in digits shows even in a small computation:

```
System.out.println(1.1f * 3);   // 3.3000002
System.out.println(1.1 * 3);    // 3.3000000000000003
```

Neither result is exactly 3.3. The error in the `float` shows in the seventh digit, and the error in the `double` in the sixteenth. The source of the error is the number itself: 1.1 cannot be written as a finite fraction in binary, so it is stored rounded to the nearest representable value.

2.2.2.7 char and boolean

`char` holds a single character. It occupies 2 bytes (16 bits) in memory and is **unsigned**: its range is 0 to 65,535. Each value is a Unicode character code. Unicode is an international standard that gives a number to the characters of the world's writing systems. `char` literals are written in single quotes: `'A'`, `'ş'`, `'7'`. The difference between single and double quotes matters: `'A'` is a `char`, while `"A"` is a `String` of one character.

Knowing the code of a character makes it possible to compute with characters. Three codes are used often: `'A'` = 65, `'a'` = 97, `'0'` = 48. The codes of the letters and digits of the English alphabet are the same as those in the ASCII table and are consecutive: `'B'` = 66, `'C'` = 67, `'1'` = 49. The code of the character `'7'` is not 7 but 55. The digit character and the numeric value of the digit are different things.

`boolean` holds a logical value. It has only two values: `true` and `false`. The information it carries is one bit; how many bytes it takes in memory is determined by the JVM. Unlike in C, a `boolean` in Java is not a number. `boolean done = 0;` does not compile, and `if (1)` cannot be written. `boolean` values are used in conditions and for the results of comparisons:

```
boolean passed = score >= 50;
```

In this line the result of the comparison `score >= 50`, that is, `true` or `false`, is written into the variable `passed`. If `score` is 72, `passed` becomes `true`.

```
char letter = 'A';
boolean valid = true;
System.out.println(letter);
System.out.println(valid);
System.out.println(!valid);
```

```
A
true
false
```

When `println` is given a `char`, it writes the character itself: `A`. It does not write its code, 65. `boolean` values are written as the words `true` and `false`. The `!` sign on the third line is the logical NOT operation and turns `true` into `false`.

2.2.2.8 Seeing the limits with a program

Instead of memorizing the tables, you can print the limits with a program. Every primitive type has a **wrapper class**: `Byte` for `byte`, `Integer` for `int`, `Double` for `double`, and `Character` for `char`. These classes provide the limits of the type through the constants `MIN_VALUE` and `MAX_VALUE`.

```
System.out.println("byte  : " + Byte.MIN_VALUE + " .. " + Byte.MAX_VALUE);
System.out.println("short : " + Short.MIN_VALUE + " .. " + Short.MAX_VALUE);
System.out.println("int   : " + Integer.MIN_VALUE + " .. " + Integer.MAX_VALUE);
System.out.println("long  : " + Long.MIN_VALUE + " .. " + Long.MAX_VALUE);
System.out.println("float : " + Float.MIN_VALUE + " .. " + Float.MAX_VALUE);
System.out.println("double: " + Double.MIN_VALUE + " .. " + Double.MAX_VALUE);
System.out.println("char  : " + (int) Character.MIN_VALUE + " .. "
                              + (int) Character.MAX_VALUE);
```

```
byte  : -128 .. 127
short : -32768 .. 32767
int   : -2147483648 .. 2147483647
long  : -9223372036854775808 .. 9223372036854775807
float : 1.4E-45 .. 3.4028235E38
double: 4.9E-324 .. 1.7976931348623157E308
char  : 0 .. 65535
```

`4.9E-324` in the output is scientific notation and means 4.9×10^{-324}. The letter `E` is read as “times ten to the power.” In the `float` and `double` lines, the value on the left is not the smallest *negative* number but the smallest number greater than zero. This differs from the integer types and is often

confused. The `(int)` cast is used on the `char` line. Without the cast, `println` would try to write these two values as characters. Code 0 is an invisible control character, and 65,535 is not a printable character either.

2.2.2.9 Overflow

Definition 2.2.9

Overflow occurs when the result of an operation does not fit in the range of values of its type. In Java, integer overflow raises no error: the value **wraps around** to the other end of the range.

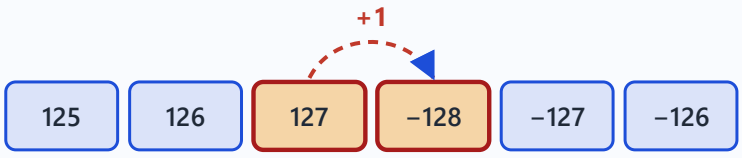
Overflow is easy to picture with a clock. If the hand is at 12 and moves one hour forward, it goes to 1, not 13. Integer types behave the same way: adding 1 to the largest value returns to the smallest. What matters is that this happens silently. The program gives neither a compile-time error nor a run-time error; it carries on with the wrong result.

```
int max = Integer.MAX_VALUE;
System.out.println(max);
System.out.println(max + 1);   // overflow!
byte n1 = 80;
byte n2 = 50;
byte sum = (byte) (n1 + n2);
System.out.println(sum);        // not 130
long fixed = (long) max + 1;
System.out.println(fixed);
```

```
2147483647
-2147483648
-126
2147483648
```

The lines run as follows:

- `max` holds the largest value of `int`, 2,147,483,647, and this value is printed on the first line.
- `max + 1` is an addition between two `int` values, and its result is an `int` too. The result does not fit and wraps around to the smallest value: `-2147483648`.
- `n1 + n2` gives 80 + 50 = 130. 130 exceeds 127, the limit of `byte`. The `(byte)` cast takes the last 8 bits of the result, and the value becomes 130 − 256 = −126.
- In the expression `(long) max + 1` the cast is done before the addition. The operation proceeds in `long`; the 64-bit range holds 2,147,483,648 comfortably.



byte: adding 1 to the maximum “wraps” to the minimum

Figure 2.2.14: Overflow. When 1 is added to 127 in type `byte`, the value becomes −128; the number line closes up like a circle.

The fix is to use a wider type. But where the cast goes matters. The line `long wrong = max + 1;` still gives −2147483648. The right-hand side is computed between two `int` values and overflows first; only the overflowed result is assigned to the `long` variable. The conversion must be done before the operation: `(long) max + 1` or `max + 1L`.

Example 2.2.1

What comes out if the square of one million is computed with `int`? With `int million = 1_000_000;`, the true value of `million * million` is 10^{12}, which is far beyond the limit of about 2.1 billion for `int`. The result Java gives is `-727379968`. This value is the signed reading of the last 32 bits of the binary representation of 10^{12}. When the same computation is written `(long) million * million`, the correct result, `1000000000000`, is obtained. Here the negative result gives the overflow away. Not every overflow is this obvious; the result may be positive but wrong.

Let us follow step by step how overflow arises at the bit level, using a `byte` counter:

```
byte b = 125;
for (int i = 1; i <= 4; i++) {
    b++;
    System.out.println(b);
}
```

`b` starts at 125 and the loop runs four times. Table 2.2.6 gives the value and the binary representation in each pass.

`i`	`i <= 4`	After `b++`	Binary	Output
—	—	125 (start)	`01111101`	
1	true	126	`01111110`	126
2	true	127	`01111111`	127
3	true	**−128**	`10000000`	-128
4	true	−127	`10000001`	-127

`i`	`i <= 4`	After `b++`	Binary	Output
5	false	—	—	loop ends

Table 2.2.6: Trace of the `byte` counter. The leftmost bit is the sign bit and has weight −128.

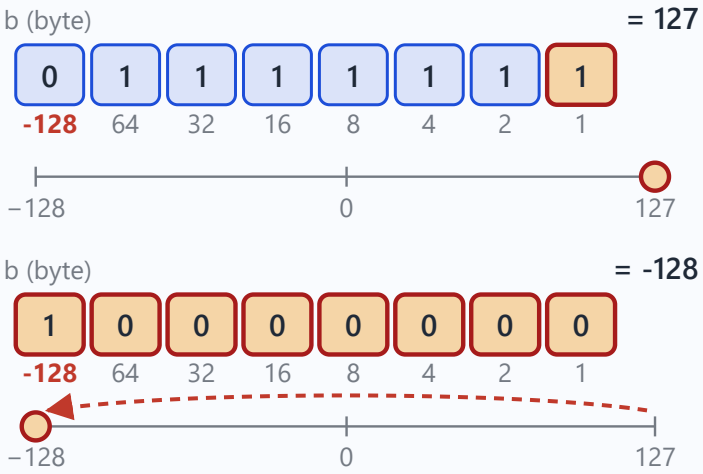


Figure 2.2.15: The moment of overflow. (a) `b` = 127, binary `01111111`: the largest value of `byte`. (b) Adding 1 gives `10000000` in binary addition; the sign bit becomes 1 and the value wraps to −128.

The cause of overflow is binary addition itself. The binary representation of 127 is `01111111`. When 1 is added, all the 1s turn to zero and the carry moves into the leftmost bit: `10000000`. This bit is the sign bit, and its weight is −128, not +128. Since the other bits are zero, the value is −128. This representation is called **two's complement**. It was chosen because the processor can use the same addition circuit for both positive and negative numbers. The program prints 126, 127, −128, −127 and gives no warning.

The `b++` line in the table needs a note too. If `b = b + 1;` had been written, the code would not compile, because `b + 1` is an `int` expression and can be assigned to a `byte` only with an explicit cast. The `++` and compound assignment operators perform the conversion themselves. Overflow is also a consequence of this hidden conversion.

2.2.2.10 Primitive types and reference types

Definition 2.2.10

A variable of a **primitive type** stores the value itself. A variable of a **reference type** points to an object: the variable holds not the object itself but its location in memory, that is, a **reference** to the object.

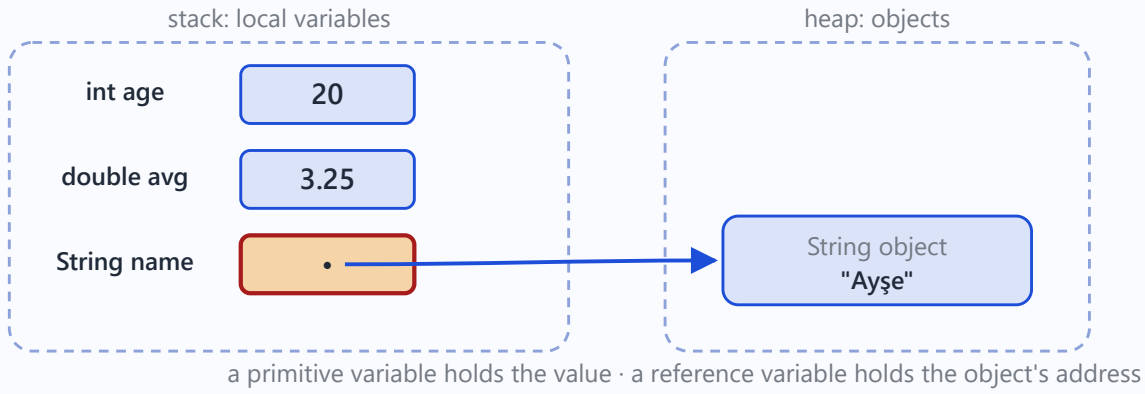


Figure 2.2.16: Primitive and reference types. `int` and `double` variables hold the value directly; a `String` variable refers with an arrow to an object in heap memory.

There are eight primitive types, and their names begin with a lowercase letter. All the remaining types are reference types. `String` (a sequence of characters), arrays, classes, interfaces, and `enum`s are in this group. The names of reference types begin with an uppercase letter. Objects are usually created with `new`. `String` is a common exception to this rule: a string literal such as `"Ali"` creates an object even without `new`.

A reference that points to no object has the value `null`. `null` must not be confused with the empty string. The empty string `""` is a real object and has length zero: `"".length()` gives 0. `null` means "there is no object here." If a method is called on a `null` reference, the program stops with `NullPointerException`. For a local `String` variable three situations are distinct. Code that prints a variable to which no value has been assigned does not compile. A variable assigned `null` prints `null`. A variable assigned `""` prints an empty line.

The difference between the two groups of types shows in assignment too. After the lines `int a = 5;` `int b = a;`, `b` holds a separate copy of 5; when `b` changes, `a` is not affected. Two reference variables, however, can point to the same object. This distinction is one of the fundamental topics of object-oriented programming.

2.2.3 Identifiers and keywords

Each of the variables, methods, and classes in a program has a name. The programmer is free to choose these names, but the limits of that freedom are clear. Some characters cannot be used in names, and some words are reserved for the language. There are also writing habits that the Java community follows. This section takes up the three levels in turn: the compiler's mandatory rules, reserved words, and conventions.

2.2.3.1 Identifier rules

> **Definition 2.2.11**
>
> An **identifier** is the name given to variables, classes, methods, and other program elements.

Identifiers follow these rules:

- They may contain only **letters**, **digits** (0–9), the underscore `_`, and the dollar sign `$`. Characters such as spaces, `-`, `@`, and `.` cannot appear.
- The first character must be a letter, `_`, or `$`. An identifier **cannot start with a digit**.
- Keywords and the literals `true`, `false`, and `null` cannot be identifiers.
- There is no length limit.
- Uppercase and lowercase are distinct: `personName` and `personname` are different variables. A name written differently from its declaration gives a `cannot find symbol` error.

There is a reason for the ban on starting with a digit. When the compiler sees a digit, it starts reading a number. Meeting `2count`, the compiler first reads the number 2 and then treats the `count` that follows as a separate word. For the same reason `first-last` is read as "first minus last," because `-` is the subtraction operator.

Java accepts as letters not only English letters but all Unicode letters. A variable named `öğrenci` ("student" in Turkish) compiles and runs. Even so, it is recommended to use only ASCII letters in names. When files are opened with a different character encoding, accented letters can be corrupted, and the code may stop compiling on other systems. That is why names are written `studentNo` and `counter`, not with accented letters. Valid examples: `userName`, `totalScore`, `student_name`, `data1`, `_counter`.

2.2.3.2 Valid and invalid names

Name	Status	Why?
`studentNo`	✓	starts with a letter
`num_2`	✓	letters, digits, and `_`
`$price`	✓	`$` is allowed, but not recommended
`2count`	✗	starts with a digit
`salary calc`	✗	contains a space
`first-last`	✗	`-` is the subtraction operator
`@name`	✗	`@` is not allowed
`class`	✗	a keyword

Name	Status	Why?
`Class`	✓	capital C: not a keyword

Table 2.2.7: Examples of identifiers. ✓ valid, ✗ invalid.

The `$` sign is valid but is used in class names the compiler itself generates. That is why programmers are advised not to use it. `Class` compiles, because Java is case-sensitive and all keywords are lowercase. But `Class` is the name of an existing class in the Java API and would surprise the reader.

An invalid name does not always produce a clear message. A name that starts with a digit confuses the compiler; the message points at the statement, not at the name:

```
int 2count = 20;
```

```
Ident.java:3: error: not a statement
        int 2count = 20;
        ^
Ident.java:3: error: ';' expected
        int 2count = 20;
           ^
2 errors
```

After the word `int` the compiler expects a name. On seeing `2`, it realizes there is no name here and treats `int` on its own as a meaningless statement (`not a statement`). It then tries to read the `2count` part as the beginning of a new statement and expects a semicolon. Neither message says what the real problem is, namely that the name starts with a digit. When messages look confusing, the first place to look is whether the names on the line follow the rules.

2.2.3.3 Keywords

Definition 2.2.12

A **keyword** (also **reserved word**) is a word that has a specific job in the language and is therefore reserved. Keywords cannot be used as variable, class, or method names. All of them are written in lowercase.

Java 21 has 51 keywords. Table 2.2.8 groups them by function.

Group	Keywords	Count
Types	`boolean byte char short int long float double void`	9
Control flow	`if else switch case default for while do break continue return`	11

Group	Keywords	Count
Classes and objects	`class interface enum extends implements new this super instanceof package import`	11
Modifiers	`public protected private static final abstract native synchronized volatile transient strictfp`	11
Exceptions	`try catch finally throw throws assert`	6
Reserved, unused	`const goto _`	3

Table 2.2.8: The 51 keywords of Java 21, by functional group.

Three words are often mistaken for keywords: `true`, `false`, and `null`. These are not keywords but **literals**. Even so, they cannot be used as names either. Another group is the **contextual keywords**. Words such as `var` (since Java 10), `yield` (since Java 14), and `record` (since Java 16) carry a special meaning only in certain places. Because they were added to the language later, they may have been used as variable names in old programs. To keep that code from breaking, they were not made full keywords. `String`, `System`, and `main` do not belong to any group; they are the names of classes and methods in the API. The line `int String = 3;` therefore compiles, but it should never be written, because it would surprise the reader.

When a keyword is used as a name, the compiler expects the construct that the word starts. `new` starts an object-creation expression; a class name and parentheses must follow it:

```
int new = 5;
```

```
Keyword.java:3: error: not a statement
        int new = 5;
        ^
... (4 errors in total)
```

A single faulty word produces four messages. After `new`, the compiler expects a class name (`<identifier> expected`) and then `(` or `[`. None of the messages says "`new` is a keyword." As with identifier errors, a sequence of messages that looks meaningless often indicates that there is a problem with the name itself.

2.2.3.4 Groups of keywords

Most keywords come into their full meaning only when the related constructs are written. At the start it is enough to know briefly what each does. The type keywords are the names of the primitive types; `void` is added to them. `void` is not a type but a marker saying that a method returns no value: `public static void main`.

Word	Purpose
`if` / `else`	Checks the condition; if true the `if` block runs, otherwise the `else` block.
`switch`	Performs different operations for different values of a variable.
`case`	Represents one case (value) inside a `switch`.
`default`	The operation to perform inside a `switch` when no case matches.
`for`	Repeats an operation with a counter as long as the condition holds.
`while`	Repeats the operation as long as the condition holds.
`do`	The `do-while` loop: the body runs at least once, and the condition is checked at the end.
`break`	Leaves the loop or the `switch` block at once.
`continue`	Skips the rest of this pass of the loop and moves to the next one.
`return`	Ends the method; returns a result if needed.

Table 2.2.9: Control-flow keywords.

Word	Purpose
`class`	Defines a class. Classes are templates for objects and the basic building blocks of Java programs.
`interface`	Defines an interface: a contract that specifies the methods a class must provide.
`enum`	A type made of fixed values: `enum Color { RED, GREEN }`.
`extends`	Inheritance: a class taking over the fields and methods of another class.
`implements`	A class implementing an interface, that is, providing the interface's methods.
`new`	Creates an object: sets aside memory and gives initial values.
`this`	A reference to the object itself; tells a field apart from a local variable of the same name.
`super`	Accesses the methods, fields, and constructor of the superclass.
`instanceof`	Checks whether an object is of a given class or interface type.
`package` / `import`	`package` sets the package (namespace) of the class; `import` makes classes in other packages available.

Table 2.2.10: Keywords for classes and objects.

Word	Purpose
`public`	Accessible from everywhere, from all classes.
`protected`	Accessible from the same package and from subclasses.
`private`	Accessible only from inside the same class.
`static`	The member belongs to the class, not to an object; it is shared among all objects.
`final`	Variable: its value cannot change. Method: cannot be overridden in a subclass. Class: cannot be inherited.
`abstract`	An abstract class (cannot be instantiated) or a method without a body; a subclass must write the body of the method.
`native`	A method whose body is written outside Java, in platform-specific code (C/C++); it is linked with JNI.

Word	Purpose
`synchronized`	In a multithreaded program, lets only one thread at a time enter a block.
`volatile`	Makes sure a variable is always read with its current value in every thread.
`transient`	This field is ignored when an object is serialized (saved).
`strictfp`	Makes floating-point operations give the same result on every platform. Since Java 17 this is the default anyway; the word no longer has any effect.

Table 2.2.11: Modifiers.

The words in the exception group concern errors that occur while the program runs. `try` surrounds code that may fail. `catch` catches and handles the exception raised in that block. The `finally` block runs at the very end whether or not an exception occurred; closing a file that was opened is written in this block. `throw` throws an exception object and interrupts the normal flow. `throws` lists in a method header the exception types the method may throw. `assert` checks that a condition is true. This check is off by default and is turned on with `java -ea`.

The last group is interesting. `const` and `goto` are reserved words that have no function. They were reserved so that programmers coming from C and C++ would not use them by mistake. A lone underscore `_` has not been allowed as a name since Java 9. When `int _ = 3;` is written in Java 21, the compiler reports that this use is a preview feature and is turned off. With Java 22, `_` definitively acquired the meaning “unused variable.”

2.2.3.5 Naming conventions

Besides the compiler’s mandatory rules, there are writing habits followed by the Java community. These are called **naming conventions**.

What?	Convention	Examples
Classes and interfaces	PascalCase, starts with an uppercase letter	`String`, `Student`, `HelloWorld`
Variables, fields, methods	camelCase, starts with a lowercase letter	`firstName`, `classAverage`, `calculate()`
Primitive types	lowercase (they are keywords)	`int`, `float`, `boolean`
Named constants	UPPER_CASE, words separated by `_`	`PI`, `DAYS_PER_WEEK`, `MAX_COUNT`
Packages	all lowercase	`syntax`, `tr.edu.giresun`

Table 2.2.12: Java naming conventions.

In **camelCase** the first word is lowercase and the following words begin with an uppercase letter. The capital letters are likened to a camel’s humps, hence the name. In **PascalCase**, every word, including the first, begins with an uppercase letter. This distinction gives the reader information at once: `Student` is a class, `student` is a variable, and `STUDENT_COUNT` is a constant.

```
class StudentRecord { }    // class
String fullName;       // variable
boolean isMarried;           // yes/no
final int MAX_COUNT = 100; // constant
```

These are not compiler rules but **conventions**. If they are not followed, the code still compiles. `class student_record` or `int TotalScore` runs without error. But it becomes harder for another Java programmer to read such code. Conventions are the common language of teamwork. The Java API itself follows these rules: `String` is a class, `length()` is a method, and `Integer.MAX_VALUE` is a constant.

2.2.4 Assignment, literals, and conversion

After variables are declared they receive values, and those values change. This section explains how a value is put into a variable, how values written directly in code are read, and how a value moves from one type to another. Type conversion is one of the topics that causes the most errors in beginning Java programs.

2.2.4.1 Assignment statements

Definition 2.2.13

Assignment gives a value to a variable. It is written with the assignment operator `=`: `variable = expression;`. First the expression on the right is computed, and then the result is written into the variable on the left.

The expression on the right can be a constant value, another variable, or an operation:

```
int num, result;
int num1 = 4, num2 = 6;
String name;

num = 10;             // literal
result = num1 + num2; // sum: 10
name = "Ahmet";        // text
num = num * 2;       // 10 → 20
```

The first three lines are declarations. `num1` and `num2` receive values together with their declarations; `num`, `result`, and `name` are still empty. Then come four assignments, each with a different right-hand side. The first is a constant (10), and the second is the sum of two variables (4 + 6 = 10). The

third is a piece of text (`"Ahmet"`), and the fourth is an operation that uses the variable's own value. After the last line `num` is 20.

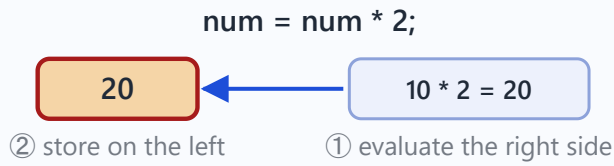


Figure 2.2.17: `num = num * 2` : first the right-hand side 10 * 2 = 20 is computed, then the result is written into the `num` box.

The `=` sign is not the equality of mathematics. In mathematics *x = x + 1* is a contradictory equation; no number can equal one more than itself. In Java, `x = x + 1` means "let the new value of `x` be one more than its old value." Reading "gets" instead of "equals" prevents this misunderstanding: "`x` gets `x + 1` ."

Only a variable can appear on the left side of an assignment. The line `5 = num;` does not compile:

```
Left.java:4: error: unexpected type
        5 = num;
        ^
  required: variable
  found:    value
1 error
```

The message says that a variable was expected on the left (`required: variable`) but a value was found (`found: value`). Types must also be compatible in an assignment. A number cannot be assigned to a `String` variable, nor text to an `int` variable. The lines `int x = 3; String s = x;` give the message `incompatible types: int cannot be converted to String` .

An assignment is also an expression and produces a value: the value assigned. That is why `a = b = 0;` can be written. Assignment groups from right to left; first `b = 0` is done, and the value of that expression, 0, is assigned to `a` . Both variables are zeroed.

2.2.4.2 Compound assignment

Definition 2.2.14

Compound assignment operators combine an operation and an assignment: `+=` , `-=` , `*=` , `/=` , `%=` . The notation `x op= y` does the same job as `x = x op y` .

```
int x = 5;
int y = 10;
x = x + y;
x -= y;
x *= 2;
x /= 4;
x %= 3;
System.out.println("x = " + x + ", y = " + y);
```

The program prints `x = 2, y = 10`. Table 2.2.13 gives the values after each line.

Line	Written out	Computation	`x`	`y`
`int x = 5;`	—	—	5	
`int y = 10;`	—	—	5	10
`x = x + y;`	—	5 + 10	**15**	10
`x -= y;`	`x = x - y`	15 − 10	**5**	10
`x *= 2;`	`x = x * 2`	5 × 2	**10**	10
`x /= 4;`	`x = x / 4`	10 / 4 (integer division)	**2**	10
`x %= 3;`	`x = x % 3`	the remainder of 2 divided by 3	**2**	10

Table 2.2.13: Trace of compound assignment.



Figure 2.2.18: Three moments of compound assignment. (a) After `x = x + y`, `x` = 15. (b) After `x *= 2`, `x` = 10. (c) After `x /= 4`, `x` = 2; `y` stays 10 throughout.

Two steps need attention. In the line `x /= 4`, 10 / 4 is computed between two integers and the result is 2, not 2.5; the fractional part is discarded. In the line `x %= 3`, since 2 is smaller than 3, the quotient is 0 and the remainder is 2. `y` never changes because it is never on the left side of a line.

Compound assignment has two subtleties. First, the right-hand side is computed as a whole, and then the operation is performed. The expression `k *= 2 + 3` means `k = k * (2 + 3)`, not `k = k * 2 + 3`; if `k` is 7, the result is 35. Second, a compound assignment converts the result to the type of the variable on the left by itself. `byte b = 10; b += 5;` compiles and `b` becomes 15. But `b = b + 5;` does not compile, because `b + 5` is an `int`. This convenience has a price: `int m = 10; m /= 0.3;` compiles without error and `m` becomes 33. 10 / 0.3 = 33.33… is computed, and then the fraction is silently discarded.

2.2.4.3 Initial and default values

A variable can be declared without a value. An initial value can also be given together with the declaration: `int age = 25;`. This is called **initialization**. When the value is given depends on two different rules, according to where the variable is declared.

Local variables, that is, variables declared inside a method, receive no default value. A local variable that is read before a value has been assigned prevents the program from compiling:

```
int count;
System.out.println(count);
```

```
Init.java:4: error: variable count
  might not have been initialized
```

The compiler examines every possible path through the program and sees that the point where `count` is read can be reached without a value. This check prevents a random memory content from being used silently. In C the same mistake compiles, and the program may print a different, meaningless number each time it runs.

Class fields, by contrast, automatically receive a **default value**. Table 2.2.14 gives these values.

Field type	Default
`int`, `long`, `byte`, `short`	0
`double`, `float`	0.0
`char`	`'\u0000'` (code 0)
`boolean`	`false`
`String` and other references	`null`

Table 2.2.14: Default values of class fields.

When the unassigned `int`, `long`, `double`, `float`, `char` (its code), `boolean`, and `String` fields of a class are printed in turn, the output is `0 0 0.0 0.0 0 false null`. Still, it is a good habit not to rely on default values and to give the value explicitly. An explicitly written `int count = 0;` tells the reader that the zero is a deliberate choice.

2.2.4.4 Literals

Definition 2.2.15

A **literal** is a value written directly in source code: `2`, `3.7`, `'y'`, `"Ali"`, `true`, and so on.

Every literal has a type, and the type can be told from the way it is written:

- **Integer:** may begin with `+` or `-` but **cannot contain a comma**: `+42`, `-17`. For readability, underscores can be placed between the digits (added in Java 7): `1_000_000`. Hexadecimal numbers begin with `0x` and binary numbers with `0b`: `0x1F`, `0b1010`.
- **Floating-point:** the decimal separator is the period. Scientific notation is written with `e`: `2.5e3` = 2.5×10^3, `7.68e-2` = 7.68×10^{-2}.
- **Character:** in single quotes: `'y'`. Special characters are written with a backslash: `'\n'` newline, `'\t'` tab, `'\''` single quote, `'\\'` backslash.
- **Text:** in double quotes: `"Hello"`. **Logical:** `true`, `false`.

```
System.out.println(2.5e3);
System.out.println(7.68e-2);
System.out.println(1_000_000);
System.out.println(0x1F);
System.out.println(0b1010);
System.out.println(3000000000L);
System.out.println("Ali\tVeli\n\"Java\"");
```

```
2500.0
0.0768
1000000
31
10
3000000000
Ali     Veli
"Java"
```

The output shows several things. `2.5e3` is a `double` and is printed as `2500.0`; even though its value is whole, the trailing `.0` reveals its type. The underscores are only for readability in source code and do not appear in the output. `0x1F` is 1 × 16 + 15 = 31 in hexadecimal and `0b1010` is 8 + 2 = 10 in binary; `println` writes both in decimal. In the text on the last line, `\t` produces a tab and `\n` a line break; `\"` puts a double quote inside the text. A single `println` therefore writes two lines.

Common mistake

Carrying the decimal comma of Turkish into code. `double rate = 0,45;` does not compile; in Java the decimal separator is the period: `0.45`. Similarly, a period cannot be used as a thousands separator: `1.000.000` is meaningless. A thousand is written `1000` or `1_000`. Another trap is a leading zero: `010` is read in base eight and its value is 8, not 10.

2.2.4.5 The precision of floating-point numbers

Floating-point numbers are stored with a limited number of bits in base two. In base ten, 1/3 = 0.333… goes on forever and cannot be written exactly with a finite number of digits. A similar situation exists in base two: $1/10 = 0.000110011\ldots_2$ repeats forever. The innocent-looking 0.1 in base ten is therefore stored approximately in a computer. The exact value of 0.1 in a `double` is a number like 0.1000000000000000055511151231257827….

```
System.out.println(1.0 / 3.0);
System.out.println(1.0/3 + 1.0/3 + 1.0/3);
System.out.println(0.1 + 0.2);
System.out.println(0.1 + 0.2 == 0.3);
double t = 0.0;
for (int i = 0; i < 10; i++) t += 0.1;
System.out.println(t);
double diff = (0.1 + 0.2) - 0.3;
System.out.println(Math.abs(diff) < 1e-9);
```

```
0.3333333333333333
1.0
0.30000000000000004
false
0.9999999999999999
true
```

The results read line by line as follows:

- `1.0 / 3.0` is an approximate value and is printed with 16 digits.
- The sum of three `1.0/3` values comes out exactly `1.0`. Here the rounding errors happened to cancel each other. This is luck, not a rule.
- The result of `0.1 + 0.2` is `0.30000000000000004`. The small errors of the two numbers add up, and the result differs from the stored value of the constant `0.3`. That is why `0.1 + 0.2 == 0.3` gives `false`.
- Adding 0.1 ten times does not give exactly 1: `0.9999999999999999`. Small errors accumulate with each addition.
- On the last line the absolute value of the difference of the two numbers is compared with a very small bound, 10^{-9}. The difference is about 5.55×10^{-17}; because it is smaller than the bound, the result is `true`.

This behavior is not specific to Java. All modern processors store floating-point numbers according to the IEEE 754 standard, and C, Python, and JavaScript give the same results. Two rules follow. First, floating-point numbers are not compared with `==`; instead one checks whether the difference is very small: `Math.abs(a - b) < 1e-9`. Second, `double` is not used in money calculations that must be

correct to the last cent. There is the `BigDecimal` class for this; the sum of `new BigDecimal("0.1")` and `new BigDecimal("0.2")` is exactly 0.3. Another way is to keep amounts in cents as `long` integers.

Common mistake

Building a loop with a floating-point counter. The loop `for (double t = 0; t != 1.0; t += 0.1)` never ends. `t` never reaches exactly 1.0; after 0.9999999999999999 it jumps to 1.0999999999999999, and `t != 1.0` stays true forever. The loop counter must be an integer; if a floating-point value is needed, it is computed from the counter: `double t = i * 0.1;`.

2.2.4.6 Named constants

Definition 2.2.16

A **named constant** is a variable declared with `final` whose value cannot be changed once it has been given. The class-level pattern is: `public static final Type CONSTANT_NAME = value;`.

Numbers embedded in code with no stated meaning are called **magic numbers**. That the 7 in the line `days = 4 * 7;` is the number of days in a week can be guessed only from context. When the same line is written `days = 4 * DAYS_PER_WEEK;`, the meaning shows by itself. If the value changes, it is also updated in a single place. Constant names are written in uppercase, with words separated by underscores.

```
public class Constants {
    public static final double PI = 3.14159;
    public static final int DAYS_PER_WEEK = 7;

    public static void main(String[] args) {
        double r = 2.0;
        double area = PI * r * r;
        System.out.println("Area: " + area);
        int days = 4 * DAYS_PER_WEEK;
        System.out.println(days + " days");
    }
}
```

```
Area: 12.56636
28 days
```

`PI` and `DAYS_PER_WEEK` are declared at class level, outside `main`. Being `static` lets them be used inside `main` without creating an object. The area comes out as 3.14159 × 2 × 2 = 12.56636. Java itself offers a more precise value: `Math.PI`. With the same radius, `Math.PI * r * r` gives 12.566370614359172.

Trying to change the constant, or assigning a `double` result to a `float`, does not compile:

```
PI = 3.14;
float area = PI * r * r;
```

```
error: cannot assign a value to
  static final variable PI
error: incompatible types: possible
  lossy conversion from double to float
```

The first message shows the `final` keyword doing its job. When someone tries to change a value that must not change, the error is caught at compile time. The second message is a type-conversion problem. `PI * r * r` is a `double` expression and `float` is a narrower type. The compiler rejects the assignment with a "possible lossy conversion" message.

2.2.4.7 Assignment compatibility

A value cannot always be assigned to a variable of another type. Java's rule is simple: if the value fits without loss, the assignment is done automatically; if loss is possible, the programmer must ask for it explicitly.

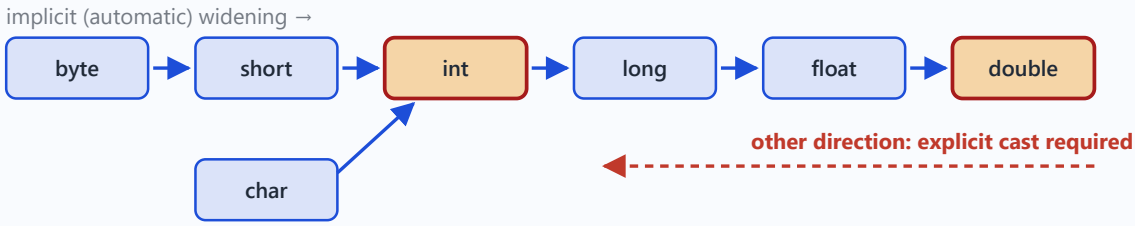


Figure 2.2.19: Assignment compatibility. In the chain `byte`, `short`, `int`, `long`, `float`, `double`, assignment to the right is automatic; a `char` can be assigned to an `int`. The reverse direction needs a cast.

Definition 2.2.17

A **widening conversion** moves a value to a type with a wider range and is done automatically. A **narrowing conversion** moves to a type with a narrower range; because a value may be lost, it requires an explicit cast.

Figure 2.2.19 shows the chain: `byte → short → int → long → float → double`. A value can be assigned directly to a variable of a type further to the right in the chain. An `int` value goes to `long`, and a `long` value to `double`, without trouble. `char` enters the chain at `int`: a `char` value can be assigned to an `int` and to the types to its right; the reverse does not compile.

```
char ch = 'A';
int n;
n = ch;   // OK: 65
ch = n;   // error
```

```
Compat.java:5: error: incompatible types:
  possible lossy conversion from int to char
        ch = n;
             ^
```

In the line `n = ch;` the code of `'A'`, 65, is written into the `int` variable. The assignment in the other direction is rejected. Even if it is known that `n` is 65 at that moment, the compiler looks at the type, not the value. An `int` variable may also hold a value that does not fit in a `char`, such as −5 or 100,000.

The chain has a subtlety. `long` is 64 bits and `float` is 32 bits; yet `long → float` is automatically allowed. The reason is that the range of `float` is much wider than that of `long`. But `float` is precise to only about 7 digits. In a very large number, the last digits are rounded. When the value `long big = 123456789123L;` is assigned to a `float` and read back, `123456790528` is obtained. The same applies to `int → float`: when 16,777,217 is assigned to a `float`, it becomes 16,777,216. Widening is safe in terms of range, but not always in terms of precision.

2.2.4.8 Implicit and explicit conversion

Definition 2.2.18

Type conversion is turning a value into another type. A conversion Java does by itself is called an **implicit conversion**; one the programmer asks for by writing the target type in parentheses is called an **explicit conversion** (a **cast**): `(int) d`.

Java converts from a smaller type to a larger type by itself. For a conversion from a larger type to a smaller one, the target type is written in parentheses. With this mark the programmer tells the compiler that the possible loss is accepted. The loss appears in two ways:

- When going from a floating-point number to an integer, the fractional part is **discarded**; the number is not rounded. `(int) 3.99` gives 3 and `(int) -3.99` gives −3. The fraction is truncated toward zero.

- An integer that does not fit overflows. `(byte) 130` takes the last 8 bits of the value and gives −126.

```
double dbl = 5;     // implicit: 5.0
System.out.println(dbl);
double d = 5.0;
int num = (int) d;   // explicit: 5
System.out.println(num);
System.out.println((int) 3.99);
System.out.println((int) -3.99);
System.out.println(Math.round(3.99));
System.out.println((byte) 130);
```

```
5.0
5
3
-3
4
-126
```

In the line `double dbl = 5;` the `int` constant 5 is implicitly converted to 5.0; that is why `println` writes `5.0`. `(int) d` is an explicit conversion and turns 5.0 into 5. There is no loss in this value, but because the compiler looks at the type, not the value, the cast is still required. `(int) 3.99` and `(int) -3.99` show that the fraction is discarded. If rounding is wanted, `Math.round` is used: `Math.round(3.99)` gives 4. `(byte) 130` shows that overflow also occurs with an explicit cast. The cast does not prevent the error; it only silences the compiler's objection.

Common mistake

Forgetting the return type of `Math.round`. `Math.round(double)` returns a `long`, not an `int`. `int n = Math.round(3.99);` therefore does not compile, with the message `possible lossy conversion from long to int`; write `int n = (int) Math.round(3.99);`. Another detail is the rounding of negative halves: `Math.round(2.5)` gives 3, but `Math.round(-2.5)` gives −2. The method adds 0.5 to the number and rounds down.

2.2.4.9 Integer division and where the cast goes

Definition 2.2.19

Integer division means that the quotient of two integers is also computed as an integer: the fractional part is discarded. **Numeric promotion** is the conversion of the operands of an operation to a common type: if one operand is `double`, the other is promoted to `double` as well.

Java decides the type of an operation by looking at the types of its operands. The type of the variable the result will be assigned to does not affect this decision. Table 2.2.15 compares six expressions written with the same numbers 7 and 2.

Expression	Result	Why?
`7 / 2`	3	two `int` values → integer division, the fraction is discarded
`7 / 2.0`	3.5	`2.0` is `double` → the operation is done in `double`
`(double) 7 / 2`	3.5	first 7 → 7.0, then the division
`(double) (7 / 2)`	3.0	first integer division (3), then 3 → 3.0: a cast that comes too late
`(int) 3.99`	3	the fractional part is discarded
`Math.round(3.99)`	4	rounds to the nearest integer (returns `long`)

Table 2.2.15: Integer division and where the cast goes.

The difference between the third and fourth rows comes from the precedence of the cast. `(double)` is a unary operator and is applied before the division. In the expression `(double) 7 / 2`, the cast applies only to 7; the division is done as `7.0 / 2`. In `(double) (7 / 2)`, on the other hand, the parentheses are computed first. Integer division gives 3, and the cast comes after the fraction is long gone.

The promotion rule has one more consequence. `byte` and `short` values are first promoted to `int` when they enter an operation. The result of `byte + byte` is therefore an `int`:

```
Sum.java:4: error: incompatible types: possible lossy conversion from int to byte
        byte c = a + b;
                   ^
1 error
```

`a` and `b` are small numbers, and even their sum of 30 would fit in a `byte`. Still, the compiler knows that the sum of two `byte` values can be as large as 254 and might not fit. The fix is to write `byte c = (byte) (a + b);` or to keep the result in an `int` variable.

To follow where the cast goes step by step, consider this code:

```
int a = 9, b = 4;
double d1 = a / b;
double d2 = (double) a / b;
double d3 = (double) (a / b);
int k = (int) 9.75;
long r = Math.round(9.75);
byte y = (byte) 200;
System.out.println(d1 + " " + d2 + " " + d3);
System.out.println(k + " " + r + " " + y);
```

The program prints two lines: `2.0 2.25 2.0` and `9 10 -56`. Table 2.2.16 shows how each variable is computed.

Variable	Expression	Intermediate steps	Value
`d1`	`a / b`	9 / 4 = 2 (integer division) → 2.0 on assignment	2.0
`d2`	`(double) a / b`	9 → 9.0; `b` is promoted to 4.0; 9.0 / 4.0	2.25
`d3`	`(double) (a / b)`	9 / 4 = 2 → 2.0	2.0
`k`	`(int) 9.75`	the fraction is discarded	9
`r`	`Math.round(9.75)`	the nearest integer (`long`)	10
`y`	`(byte) 200`	200 = `11001000` ; sign bit 1 → 200 − 256	−56

Table 2.2.16: Trace of type conversion (`a` = 9, `b` = 4).

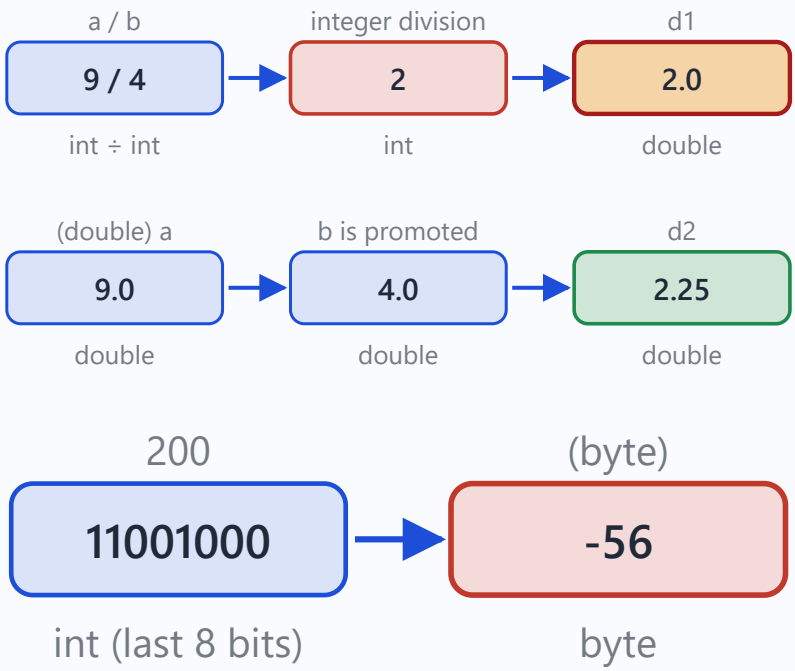


Figure 2.2.20: Where the cast goes. (a) `d1` : first the integer division gives 2, then 2 → 2.0. (b) `d2` : first 9 → 9.0, the division is done in `double` , and 2.25 results. (c) `y` : the last 8 bits of 200 are `11001000` , which read as signed is −56.

The `d1` line shows the most common mistake. That the target is `double` does not rescue the division. Java computes the right-hand side first, and at that point it does not look at where the result will be assigned. Two `int` values are divided, 2 is obtained, and only on assignment is it turned into 2.0. In the `y` line, 200 in binary is `11001000`. When these 8 bits are read as a `byte`, the leftmost 1 carries the weight −128: −128 + 64 + 8 = −56.

Example 2.2.2

The formula that converts Celsius to Fahrenheit is *F = C × 9 / 5 + 32*. With `int c = 37;`, what are the results of three different ways of writing it?

- `c * 9.0 / 5 + 32`: because 9.0 is a `double`, the computation runs in `double` from the start. 37 × 9.0 = 333.0; 333.0 / 5 = 66.6; 66.6 + 32 = **98.6**.
- `c * 9 / 5 + 32`: everything is `int`. 37 × 9 = 333; 333 / 5 = 66 (fraction discarded); 66 + 32 = **98**. The result is 0.6 too small.
- `9 / 5 * c + 32`: operations go from left to right. 9 / 5 = 1 (integer division); 1 × 37 = 37; 37 + 32 = **69**. The result is almost 30 degrees off.

In mathematics these three forms are the same. In Java the order of the operations and the types of the operands change the result. How early the integer division happens determines the size of the loss.

2.2.4.10 Converting between char and int

Every `char` is really a number: the character's Unicode code. `'A'` = 65, `'B'` = 66, `'a'` = 97. That is why arithmetic can be done on characters. The rules are the same as for the other numeric types:

- `char → int` is implicit. `(char)` is needed for `int → char`.
- If two `char` values are added with `+`, both are promoted to `int` and a number results. To get text, the expression must start with a `String`.
- In the English alphabet the codes of uppercase and lowercase letters differ by 32: `'a' - 'A'` = 97 − 65 = 32.
- The codes of the digits start at 48 and are consecutive. That is why `'7' - '0'` gives the numeric value of the digit, that is, 7.

All the character codes can be seen in the ASCII table.

```
char ch = 'A';
int code = ch;               // 65
System.out.println(code);
char next = (char) (code + 1);
System.out.println(next);
System.out.println('B' + 'C');
System.out.println("" + 'B' + 'C');
System.out.println((char) (ch + 32));
System.out.println((int) 'ş');
```

```
65
B
133
BC
a
351
```

`'B' + 'C'` gives 66 + 67 = 133; there is no text here. The expression `"" + 'B' + 'C'`, on the other hand, starts with an empty string. From left to right, first `"" + 'B'` is computed and the text `"B"` results. Then `'C'` is appended to this text: BC. `(char) (ch + 32)` turns capital A into lowercase a. The last line gives the code of the Turkish letter `'ş'`: 351. Turkish letters are not among the English letters but in a later part of the Unicode table. The rule of a difference of 32 therefore does not hold for them: `'Ş'` has code 350, `'ı'` 305, and `'İ'` 304.

Let us combine these rules in one trace:

```
char c = 'd';
int code = c;
char upper = (char) (c - 32);
int diff = 'z' - c;
char next = c;
next++;
System.out.println(code + " " + upper + " " + diff + " " + next);
System.out.println(c + 1);
System.out.println("" + c + 1);
```

Line	Computation	Result
`char c = 'd';`	the code of `'d'` is stored	`c` = `'d'` (100)
`int code = c;`	implicit `char → int`	`code` = 100
`char upper = (char) (c - 32);`	100 − 32 = 68 (`int`), with `(char)` 68 = `'D'`	`upper` = `'D'`
`int diff = 'z' - c;`	122 − 100	`diff` = 22
`char next = c; next++;`	100 → 101; `++` needs no cast	`next` = `'e'`
`println(code + " " + … )`	concatenation because of the text `" "`	`100 D 22 e`

Line	Computation	Result
`println(c + 1)`	`char` is promoted to a number: 100 + 1	`101`
`println("" + c + 1)`	first `"d"`, then `+ 1` is appended to the text	`d1`

Table 2.2.17: Trace of `char ⇔ int`.

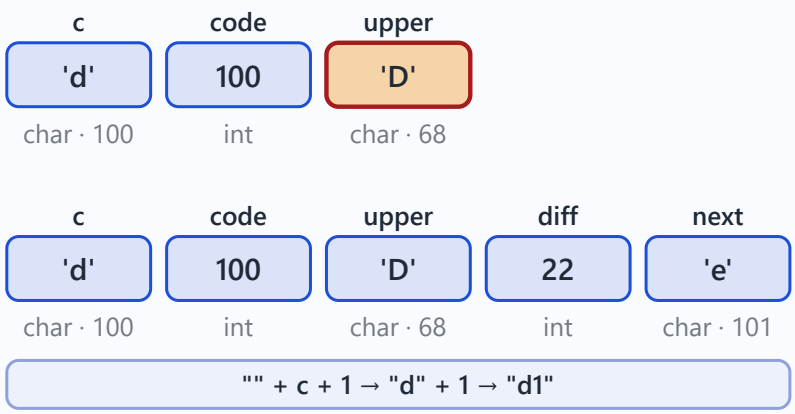


Figure 2.2.21: Two moments from the `char ⇔ int` trace. (a) `c` = `'d'`, `code` = 100, and `upper` = `'D'` have been computed. (b) All variables are ready; the last two lines print `101` and `d1`.

The last two rows of the table produce very different results from the same values. There is no text in `c + 1`. The `char` is promoted to a number and `101` is printed; no letter is written. If a letter is wanted, write `(char) (c + 1)` and `e` results. In `"" + c + 1`, on the other hand, the empty string on the left turns the whole expression into a text operation. `c` is appended to the text as `d` and `1` as `1`. The line `next++` is also worth noticing. `next = next + 1;` would not compile, because the right-hand side is an `int`. `++`, like compound assignment, does the conversion itself.

2.2.5 Operators

Operators are symbols that perform operations on values. `+`, `<`, and `&&` are **operators**; the values they work on are called **operands**. Java's operators fall into four groups: arithmetic, comparison, logical, and assignment operators. When an expression contains more than one operator, the precedence rules determine which is applied first.

2.2.5.1 Arithmetic operators

Operator	Meaning	Expression	Result
`+`	addition	`x + y`	22
`-`	subtraction	`x - y`	12
`*`	multiplication	`x * y`	85
`/`	division	`x / y`	3

Operator	Meaning	Expression	Result
`%`	remainder (mod)	`x % y`	2
		`-17 / 5`	−3
		`-17 % 5`	−2
		`17.0 / 5`	3.4

Table 2.2.18: Arithmetic operators (`x` = 17, `y` = 5).



Figure 2.2.22: When 17 objects are split into groups of 5, there are 3 groups and 2 objects are left over: `17 / 5` = 3, `17 % 5` = 2.

`/` performs integer division between two integers. `%` gives the remainder of the division. Figure 2.2.22 shows the two together: 17 = 5 × 3 + 2. The quotient and the remainder always satisfy this identity: the expression `(x / y) * y + x % y` equals `x`. With negative numbers, Java truncates the quotient toward zero: −17 / 5 is −3, not −3.4. To keep the identity, the remainder is −2: 5 × (−3) + (−2) = −17. The rule is that the sign of the remainder is the sign of the dividend. `17 % -5` therefore gives 2.

The `%` operator has a few common uses:

- **Odd-even test:** if `n % 2 == 0`, `n` is even. For an odd test, write `n % 2 != 0`. `n % 2 == 1` gives a wrong result for negative odd numbers, because `-7 % 2` is −1.
- **Last digit:** `n % 10` gives the last digit and `n / 10` gives the number with its last digit dropped. For `n` = 1234, 4 and 123 result.
- **Splitting units:** jobs such as splitting seconds into hours, minutes, and seconds are done with `/` and `%`.

Example 2.2.3

How many hours, minutes, and seconds are 3725 seconds? With `int s = 3725;`:

- `s / 3600` = 1. One hour is 3600 seconds, and it fits once in 3725.
- `s % 3600 / 60` = 125 / 60 = 2. First the 125 seconds left over after the hour are found, then the whole minutes in them are counted. `%` and `/` have equal precedence and are applied from left to right.
- `s % 60` = 5. These are the seconds left over after the whole minutes.

The result: 1 hour, 2 minutes, 5 seconds. Check: 1 × 3600 + 2 × 60 + 5 = 3725.

Java has no exponentiation operator. `2 ** 3` does not compile, and `2 ^ 3` compiles but does not raise to a power: `^` is the bitwise exclusive or, and it gives 1. For powers, `Math.pow(2, 3)` is used; the result is 8.0 of type `double`. The `+` operator has a second meaning too. If one of its operands is a `String`, it performs **concatenation**: the expression `"Age: " + 20` gives the text `Age: 20`.

In integer division, if the divisor is 0 the program stops with `ArithmeticException: / by zero`. In floating-point division no exception occurs: `5.0 / 0` gives `Infinity` and `0.0 / 0` gives `NaN` (not a number). This difference too comes from the types of the operands.

2.2.5.2 Increment and decrement

Definition 2.2.20

The **increment** (`++`) and **decrement** (`--`) operators increase or decrease a variable by 1. If the operator is written after the variable, it is the **postfix** form (`x++`); if before, the **prefix** form (`++x`).

Both `x++` and `++x` increase `x` by 1. The difference is in the **value** of the expression. The postfix form gives the old value and the prefix form gives the new value. When `i++;` is written alone as a statement, the value of the expression is not used and there is no difference between the two forms. The difference shows only when the expression is part of another operation.

```
int a = 5;
int b = a++;
System.out.println(a + " " + b);
int c = ++a;
System.out.println(a + " " + c);
int d = a--;
System.out.println(a + " " + d);
```

Line	What happens?	`a`	New variable	Output
`int a = 5;`	—	5		
`int b = a++;`	first the old value (5) is used, then `a` increases	6	`b` = 5	
`println(a + " " + b)`		6		6 5
`int c = ++a;`	first `a` increases, then the new value (7) is used	7	`c` = 7	
`println(a + " " + c)`		7		7 7
`int d = a--;`	first the old value (7), then `a` decreases	6	`d` = 7	

Line	What happens?	a	New variable	Output
`println(a + " " + d)`		6		6 7

Table 2.2.19: Trace of increment and decrement.

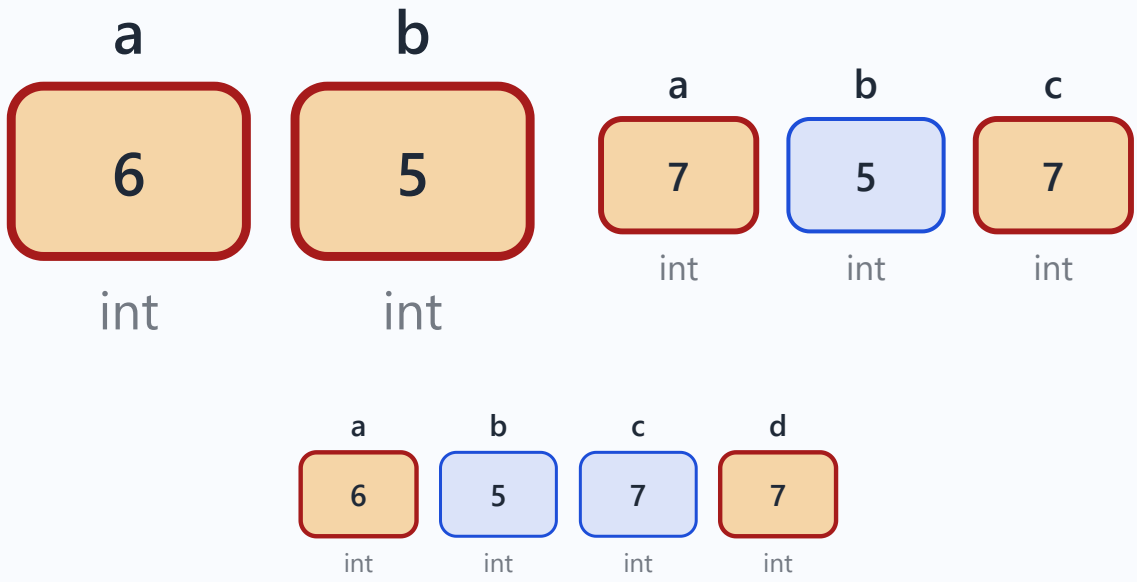


Figure 2.2.23: (a) After `b = a++`: `a` = 6, `b` = 5. (b) After `c = ++a`: `a` = 7, `c` = 7. (c) After `d = a--`: `a` = 6, `d` = 7.

Common mistake

Writing `i = i++;`. This line does not increase `i`. The right-hand side is computed first: `i++` gives the old value, say 5, and makes `i` 6. Then the assignment follows and writes that old value, 5, into `i`. The increment is thereby erased. To increase a variable, `i++;` or `i += 1;` is enough. A general rule follows from this: both changing and reading the same variable in one expression makes the code unreadable.

2.2.5.3 Operator precedence

Definition 2.2.21

Operator precedence is the order that determines which operator is applied first when an expression contains more than one. Among operators of equal precedence, **associativity** decides: left to right or right to left.

Precedence	Operators	Direction
1 (highest)	postfix: `x++` `x--`	→

Precedence	Operators	Direction
2	unary: `+x` `-x` `!` `++x` `--x` · cast `(int)`	←
3	multiplication, division, mod: `*` `/` `%`	→
4	addition, subtraction: `+` `-`	→
5	comparison: `<` `>` `<=` `>=`	→
6	equality: `==` `!=`	→
7	logical AND: `&&`	→
8	logical OR: `||`	→
9	conditional: `? :`	←
10 (lowest)	assignment: `=` `+=` `-=` `*=` …	←

Table 2.2.20: Order of precedence of the operators. → groups left to right, ← groups right to left.

The table extends the order of operations from mathematics. Multiplication and division come before addition; comparisons come after arithmetic, logical connectives after comparisons, and assignment last. This order agrees with everyday notation: the expression `x + 1 < y * 2 && z == 0` is read, even without parentheses, as expected: `((x + 1) < (y * 2)) && (z == 0)`.

For binary operators of equal precedence the one on the left is applied first: `10 - 4 - 3` = (10 − 4) − 3 = 3, not 10 − (4 − 3) = 9. For unary operators and assignment, the one on the right is applied first: `a = b = 0` zeroes `b` first and then `a`. Parentheses are used to change the order or to improve readability. A parenthesis that looks unnecessary is often useful too; it saves the reader from having to remember the precedence table.

Let us see step by step how precedence is applied to one expression: `System.out.println(2 + 3 * 4 - 6 / 2);`

1. `2 + 3 * 4 - 6 / 2`: `*` and `/` come before `+` and `-`. Of the two, the left one first: 3 * 4 = 12.
2. `2 + 12 - 6 / 2`: the remaining high-precedence operation is 6 / 2 = 3.
3. `2 + 12 - 3`: the remaining `+` and `-` have equal precedence and go from left to right: 2 + 12 = 14.
4. `14 - 3` = 11.

The result is **11**. Parentheses change the order: `(2 + 3) * 4 - 6 / 2` = 5 × 4 − 3 = 20 − 3 = 17. A few more short examples: `20 / 4 * 2` is computed from the left and gives 5 × 2 = 10, not 20 / 8. In the expression `-2 * -3` the unary minuses are applied first and the result is 6.

String concatenation obeys the same left-to-right rule and can therefore give surprising results. In the expression `1 + 2 + "3" + 4 + 5`, first 1 + 2 = 3 is computed; there is no text yet. Then 3 + `"3"` becomes text: `"33"`. Every `+` after that is concatenation: `"334"`, `"3345"`. The result is **3345**.

2.2.5.4 Comparison operators

Definition 2.2.22

Comparison operators (also relational operators) evaluate the relationship between two values. The result is always a `boolean`: `true` or `false`.

Operator	Meaning	Example	Result
`==`	equal?	`x == y`	`false`
`!=`	not equal?	`x != y`	`true`
`>`	left greater?	`y > x`	`true`
`<`	left less?	`x < y`	`true`
`>=`	greater or equal?	`x >= 30`	`false`
`<=`	less or equal?	`x <= y`	`true`

Table 2.2.21: Comparison operators (`x` = 10, `y` = 20).

The result of a comparison can be stored in a variable: `boolean bigger = y > x;`. After this line `bigger` is `true`. The spelling of the two-character operators is fixed: `>=` is correct, and `=>` cannot be written. A space cannot be put inside an operator either: `> =` does not compile.

The mathematical notation *0 < x < 10* does not work in Java. In the expression `0 < x < 10`, first `0 < x` is computed and a `boolean` results. Then this `boolean` is compared with 10, and the compiler reports an error. The correct way joins the two comparisons with logical AND: `0 < x && x < 10`.

Common mistake

Confusing `=` with `==`. `=` is assignment and `==` is comparison. When `if (x = 5)` is written, an assignment takes the place of the condition. The value of the assignment is 5, that is, an `int`. Because Java accepts only `boolean` as a condition, the code does not compile:

```
Eq.java:4: error: incompatible types: int cannot be converted to boolean
        if (x = 5) { }
              ^
1 error
```

In C the same line compiles and the condition is always taken to be true; the error stays hidden. Java's strict type checking catches this mistake at compile time. There is one exception: `boolean done; … if (done = true)` compiles, because the value of the assignment is a `boolean`. With `boolean` variables the safest way to write is not to use a comparison at all: `if (done)` and `if (!done)`.

2.2.5.5 Logical operators

Definition 2.2.23

Logical operators combine `boolean` values. AND `&&`: `true` if both conditions are `true`. OR `||`: `true` if at least one is `true`. NOT `!`: reverses the value.

A	B	A && B	A || B	!A
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Table 2.2.22: Truth table of the logical operators.

The table reads briefly as follows: `A && B` is true only when both are true; `A || B` is false only when both are false. The "or" of everyday language sometimes means "only one of the two" ("tea or coffee"). The `||` in Java, however, also gives true when both are true.

```
int x = 10, y = 20, z = 30;
System.out.println(x < y && y < z);
System.out.println(x > y || y < z);
System.out.println(!(x < y));
int age = 20;
boolean hasLicense = true;
System.out.println(age >= 18 && hasLicense);
```

```
true
true
false
true
```

In the expression `x < y && y < z` both comparisons are true (10 < 20 and 20 < 30), so the result is `true`. In `x > y || y < z` the first comparison is false and the second is true; for OR one is enough. `!(x < y)` reverses the true `x < y` and gives `false`. The last line turns a rule into code: a person who is 18 or older and holds a license may drive. Since both conditions are met, the result is `true`.

The `!` operator has high precedence. If `!x < y` is written, `!` is applied to `x` alone, and since an `int` value cannot be negated logically, the code does not compile. The negation of a comparison is therefore written with parentheses: `!(x < y)`. The same expression can be written more simply as `x >= y`.

2.2.5.6 Short-circuit evaluation

> **Definition 2.2.24**
>
> **Short-circuit evaluation** means that in the `&&` and `||` operators the right-hand side is not evaluated at all once the result is already known from the left-hand side.

In `&&`, if the left side is `false`, the result is already `false`; whatever the right side is, it cannot change that. In this case Java **does not evaluate the right side at all**. In `||`, if the left side is `true`, the result is already `true` and the right side is skipped again. In the expression `day == 6 || day == 7`, if `day` is 6 the second comparison is never made. This expression means "weekend." Its negation `!(day == 6 || day == 7)` means "weekday."

Short-circuiting does not only save time; it is also used to guard a dangerous operation. First the program checks "is the divisor nonzero?", and only if so does it divide:

```
int divisor = 0;
int num = 10;
if (divisor != 0 && num / divisor > 1) {
    System.out.println("Ratio above 1");
} else {
    System.out.println("Can't divide or too small");
}
```

```
Can't divide or too small
```

`divisor` is 0. `divisor != 0` comes out false and `&&` never moves on to the right side. The division `num / divisor` is therefore not performed, and the program goes to the `else` block. Order matters. If the condition had been written `num / divisor > 1 && divisor != 0`, the division would have been done first. The program would have stopped with a division-by-zero error, the message `java.lang.ArithmeticException: / by zero`. The guarding condition is always written on the left.

Java has two more operators that do not short-circuit: the single `&` and the single `|`. When used with `boolean` values they always evaluate both sides. The expression `a != 0 & 10 / a > 1` performs the division even when the left side is false for `a` = 0, and throws the exception. In conditions the two-character `&&` and `||` are almost always used.

2.2.5.7 The conditional operator

Definition 2.2.25

The **conditional operator** (also called the ternary operator) is a short expression that chooses one of two values according to a condition: if the condition is true the result is `value1`, otherwise `value2`. It is the only Java operator with three operands.

```
result = (condition) ? value1 : value2;
```

To decide whether a number is positive or negative:

```
int num = -5;
String result = (num > 0) ? "Positive" : "Negative";
System.out.println("Result: " + result);
```

```
Result: Negative
```

Because `num` is −5, `num > 0` is false and the value after the colon, `"Negative"`, is chosen. The `if-else` that does the same job is:

```
String result;
if (num > 0) {
    result = "Positive";
} else {
    result = "Negative";
}
```

The basic difference between the two forms is this: `if-else` is a **statement** and does a job, while `?:` is an **expression** and produces a value. That is why `?:` is used on the right of an assignment or inside a `println`. It cannot be written alone as a statement:

```
Tern.java:4: error: not a statement
        x > 0 ? System.out.println("a") : System.out.println("b");
              ^
1 error
```

The types of the two alternatives must be compatible. A few examples: `double abs = (num < 0) ? -num : num;` gives the absolute value, and for −4.5 the result is 4.5. `int bigger = (a > b) ? a : b;` chooses the larger of two numbers; for 8 and 13 the result is 13. If the alternatives are of different numeric types, the result is promoted to the wider one: the value of `true ? 1 : 2.0` is `1.0`, not 1.

Conditional operators can be nested. Whether a number is zero, positive, or negative can be found in one line:

```
int num = -5;
String result = (num == 0) ? "Zero" : (num > 0) ? "Positive" : "Negative";
System.out.println("Result: " + result);
```

`num`	`num == 0`	`num > 0`	**Result**
0	`true`	not evaluated	Zero
7	`false`	`true`	Positive
−5	`false`	`false`	Negative

Table 2.2.23: Results of a nested conditional operator.

The expression reads: “zero? yes → Zero; no → positive? yes → Positive; no → Negative.” The second `?:` is the `value2` of the first; the conditional operator groups from the right. More than two nested `?:` hurt readability. In that case an `if-else if` chain is preferred.

2.2.6 The syntax of control statements

Control statements govern the flow of a program. They provide branching on a condition and the repetition of a task. Their detailed use is a topic in itself. Here the writing rule of each and the most common syntax traps are given.

Statement	Syntax	Rule
`if` / `else`	`if (condition) { … } else { … }`	the condition must be `boolean`; a `;` after `if (…)` is an empty statement
`switch`	`switch (d) { case 1: … break; default: … }`	`int`, `char`, `String`, `enum`; without `break` execution falls through to the cases below
`switch` (Java 14 and later)	`x = switch (d) { case 1 -> "P"; default -> "?"; };`	no fall-through; an expression that produces a value
`for`	`for (int i = 0; i < n; i++) { … }`	initialization once; the condition before each pass, the update after
`while`	`while (condition) { … }`	the body may never run
`do-while`	`do { … } while (condition);`	the body runs at least once; the final `;` is required

Statement	Syntax	Rule
`break` / `continue`	`break;` · `continue;`	`break` exits, `continue` skips the rest of the pass

Table 2.2.24: The syntax of control statements.

In the `if` statement the condition is always written in parentheses and must be `boolean`. If the body is a single statement, the braces may be left out, but this habit invites mistakes. Only the first statement is attached to an `if` without braces; a second line written below it with indentation runs regardless of the condition. The compiler does not look at indentation. It is therefore safe to use braces even when the body is one line.

In the `switch` statement the value being tested can be `int` (and `byte`, `short`), `char`, `String`, or `enum`. `long`, `double`, and `boolean` cannot be used; the compiler gives a message such as `selector type double is not allowed`. In the classic form, `break` is written at the end of each `case`. If `break` is forgotten, the cases after the matching one run in turn as well. This behavior is called **fall-through**. Fall-through is sometimes used deliberately: writing `case "Sat": case "Sun":` ties two cases to one piece of code. In the arrow (`->`) form introduced in Java 14 there is no fall-through, and `switch` can produce a value. The two forms cannot be mixed in the same `switch` (`different case kinds used in the switch`).

The header of the `for` loop consists of three parts separated by semicolons. The initialization part runs only once, on entering the loop. The condition is checked before every pass; if it is false the loop ends. The update runs at the end of every pass. Every `for` loop can also be written as a `while` loop: the initialization moves before the loop and the update to the end of the body.

The difference between `while` and `do-while` is where the condition is. `while` checks the condition first; if the condition is false from the start, the body never runs. `do-while` runs the body first and checks the condition afterwards; the body runs at least once. The semicolon at the end of `do-while` is required, and if it is forgotten a `';' expected` error results.

`break` leaves the nearest enclosing loop or `switch` at once. `continue` skips only the rest of that pass and moves to the next one. In a `for` loop the update part still runs after `continue`. If a `switch` is inside a loop, a `break` inside the `switch` leaves only the `switch`; the loop continues.

Example 2.2.4

The sum of the numbers from 1 to 100 that are not multiples of 3 can be found with `continue` as follows:

```
int sum = 0;
for (int i = 1; i <= 100; i++) {
    if (i % 3 == 0) continue;   // skip multiples of 3
    sum += i;
}
```

For an `i` that is a multiple of 3, `continue` runs, `sum += i` is skipped, and the loop goes on with `i++`. Check: the sum of the numbers from 1 to 100 is 5050. The multiples of 3 are 3, 6, ..., 99; there are 33 of them and their sum is 3 × (1 + 2 + ... + 33) = 3 × 561 = 1683. The result is 5050 − 1683 = **3367**.

Common mistake

An infinite loop. A loop with no step that makes the condition false never ends. In the loop `for (int i = 1; i <= 10; i--)`, as `i` decreases the condition stays true. In `int i = 1; while (i <= 3) { System.out.println(i); }`, `i` is never updated in the body and 1 is printed on the screen endlessly. Whenever a loop is written, the question to ask is: "at which step will the condition become false?"

2.2.7 Common mistakes

Syntax errors fall into two groups. Those in the first group are caught by the compiler; the program never runs and the message points to the place of the error. Those in the second group compile but run wrongly. These are more dangerous, because they are noticed only when the output is examined carefully.

Mistake	Example	Message	Correct form
Missing semicolon	`System.out.println("a")`	`';' expected`	`;` at the end of the line; the error is often at the end of the previous line
File name ≠ class name	`public class HelloWorld` inside `helloworld.java`	`class HelloWorld is public, should be declared in a file named HelloWorld.java`	save the file under the class name
Uppercase versus lowercase	`string`, `Int`, `system.out`	`cannot find symbol`; `package system does not exist`	`String`, `int`, `System`
Local variable without a value	`int count; println(count);`	`variable count might not have been initialized`	give a value before reading
Literal without a suffix	`float f = 3.14;` · `long l = 3000000000;`	`possible lossy conversion from double to float` · `integer number too large`	`3.14f` · `3000000000L`
Forgetting the cast when narrowing	`char k = i;` · `int n = d;`	`incompatible types: possible lossy conversion`	`(char) i` · `(int) d`
`=` versus `==`	`if (x = 5)`	`int cannot be converted to boolean`	`if (x == 5)`

Table 2.2.25: Errors the compiler catches and their messages.

Mistake	Example	Result	Correct form
Integer division	`double avg = (3 + 4) / 2;`	3.0	`(3 + 4) / 2.0` → 3.5
A cast that comes too late	`(double) (a / b)`	the fraction is already lost	`(double) a / b`
Overflow	`long l = Integer.MAX_VALUE + 1;`	−2147483648	`Integer.MAX_VALUE + 1L`
No `break` in a `switch`	`case 1: … case 2: …`	the following cases run too	`break` at the end of every `case`, or the `->` form
`;` after `if`	`if (x > 0); { … }`	the empty statement is attached to the condition, and the block always runs	delete the `;`
Infinite loop	`for (int i = 1; i <= 10; i--)`	the loop never ends	an update that makes the condition false
`==` on floating-point values	`0.1 + 0.2 == 0.3`	`false`	`Math.abs(a - b) < 1e-9`
Order of concatenation	`"Total: " + a + b`	the numbers are written side by side	`"Total: " + (a + b)`

Table 2.2.26: Code that compiles but runs wrongly.

The `if (x > 0); { … }` mistake is especially insidious. The semicolon right after the `if` is an **empty statement** that does nothing. That is what gets attached to the condition. The braced block below it is an ordinary block independent of the condition and always runs. The block runs even when `x` = −3. The compiler does not count this as an error, because each of the two lines is valid by itself.

2.2.8 Summary

- **Compiling:** `javac Name.java` checks the source code and writes it as bytecode to `Name.class`. `java Name` starts the JVM, loads the class, and calls `main`. The same bytecode runs on every platform that has a JVM.
- **The JVM** is a stack machine: instructions push values onto the operand stack and take them from it. The interpreter runs the bytecode, and the JIT compiler turns frequently executed code into machine code.
- **Syntax:** statements end with `;`, blocks are opened and closed with `{ }`, and Java is case-sensitive. Comments are written with `//` and `/* */`.
- **Primitive types:** `byte` 1, `short` 2, `int` 4, `long` 8 bytes; `float` 4, `double` 8 bytes; `char` 2 bytes; `boolean`. An integer that exceeds the limit overflows without error and wraps to the other end of the range.
- **Floating-point numbers** are stored approximately in binary: `0.1 + 0.2` is not 0.3. Floating-point numbers are not compared with `==`.

- **Names** contain letters, digits, `_`, and `$`; they do not start with a digit; they cannot be keywords. Classes are written in PascalCase, variables in camelCase, and constants in UPPER_CASE.
- **Assignment:** the right-hand side is computed first. Compound assignments such as `+=` do the conversion themselves. A local variable must be given a value before it is read. A `final` variable receives a value once.
- **Conversion:** the direction `byte → short → int → long → float → double` is automatic. The reverse is done with `(type)`; the fraction is discarded and a value that does not fit overflows. `int / int` is integer division, and the cast must be done before the division.
- **Operators:** the order of precedence is unary > `* / %` > `+ -` > comparison > `&&` > `||` > `?:` > assignment. `&&` and `||` short-circuit. `?:` is an expression that produces a value.
- **Flow:** `if/else`, `switch` (+ `break`), `for`, `while` (0 or more passes), `do-while` (at least 1 pass). `break` exits, `continue` skips the rest of the pass.

2.2.9 Key terms

source code comment block Java Development Kit (JDK) bytecode Java Virtual Machine (JVM) JIT compiler platform independence variable declaration initialization data type primitive type reference type integer character boolean overflow identifier keyword naming convention assignment compound assignment default value literal named constant type conversion cast widening / narrowing conversion integer division expression operator increment / decrement operator precedence comparison operator logical operator short-circuit evaluation conditional (ternary) operator

---

2.2.10 Test Yourself

1. A file named `Demo.java` defines `public class Test`. The file is to be compiled with `javac Demo.java`. What happens? How can the program be made to run?

   Answer

   The compilation fails: `class Test is public, should be declared in a file named Test.java`. The name of a public class must be exactly the same as the file name. Save the file as `Test.java`, compile it with `javac Test.java`, and run it with `java Test` (without the extension).

2. When the lines `int a = 5, b = 7; int c = a * b;` are compiled, which values are on the JVM's operand stack while `a * b` is computed? What is left on the stack right after the `imul`

instruction?

Answer

First `iload_1` pushes 5 and then `iload_2` pushes 7; the stack holds 5, 7 from bottom to top. `imul` pops these two values, multiplies them, and pushes the result: only 35 remains on the stack. Then `istore_3` writes 35 into `c` and the stack is empty. The same pattern holds as for `iadd` in the addition.

3. Why does the following line give a wrong result? How is the correct value obtained?
`long microseconds = 24 * 60 * 60 * 1000 * 1000;`

Answer

All the constants are `int` and the multiplication is done in `int`. The true result is 86,400,000,000, which exceeds the `int` limit; the overflowed value is `500654080`. Only this overflowed value is assigned to the `long` variable. When `L` is added to the first constant, the multiplication runs in `long` from the start: `24L * 60 * 60 * 1000 * 1000` = `86400000000`.

4. What is `b` after the lines `byte b = 120; b += 10;`? What would happen if the same thing were written `b = b + 10;`?

Answer

`b` becomes −126. 130 exceeds the `byte` limit of 127; the compound assignment converts the result to `byte` itself, and 130 − 256 = −126 remains. `b = b + 10;`, on the other hand, does not compile: `b + 10` is an `int` expression and can be assigned to a `byte` only with `(byte)` (`possible lossy conversion from int to byte`).

5. Which of these names are valid identifiers? `_2x`, `x-2`, `int`, `Integer`, `total$`, `9th`

Answer

Valid: `_2x` (starts with an underscore), `Integer` (the name of a class, not a keyword; but using it as a name would surprise the reader), `total$` (`$` is allowed). Invalid: `x-2` (`-` is the subtraction operator), `int` (a keyword), `9th` (starts with a digit).

6. With `char c = '5';`, what do `System.out.println(c + 1);` and `System.out.println(c - '0' + 1);` print?

Answer

The first line prints 54: the code of `'5'` is 53, and `c + 1` is an `int` sum. The second line prints 6: `c - '0'` = 53 − 48 = 5, that is, the numeric value of the digit; 1 is added to it.

7. After the lines `int x = 7; int y = x-- - --x;`, what are `x` and `y`?

   Answer

   The operands are evaluated from left to right. `x--` gives the old value, 7, and `x` becomes 6. `--x` first makes `x` 5 and gives 5. `y` = 7 − 5 = 2, and `x` = 5.

8. With `int a = 0;`, what is the difference between the conditions `if (a != 0 & 10 / a > 1)` and `if (a != 0 && 10 / a > 1)`?

   Answer

   The single `&` does not short-circuit and evaluates both sides. Even though the left side is false, the division `10 / a` is performed and the program stops with `ArithmeticException: / by zero`. `&&`, by contrast, never looks at the right side once the left side is false; the condition counts as false and the program continues safely.

9. What is `k` at the end of the two loops below, each starting with `int k = 3;`?
   `do { k -= 4; } while (k > 5);`
   `while (k > 5) { k -= 4; }` (again starting with `k` = 3)

   Answer

   `do-while` runs the body once: `k` becomes −1, `-1 > 5` is false, and the loop ends; the result is −1. `while` checks the condition first: because `3 > 5` is false, the body never runs; the result is 3.

2.2.11 Further reading

- Oracle, *The Java Tutorials*, "Language Basics" section: the official tutorial that explains variables, operators, expressions, and control-flow statements with short examples.
- J. Gosling et al., *The Java Language Specification*, Java SE 21: the exact definition of keywords, literals (Chapter 3), types (Chapter 4), and the conversion rules (Chapter 5). The place to turn when you wonder why a compiler message appears.
- T. Lindholm et al., *The Java Virtual Machine Specification*, Java SE 21: defines the bytecode instructions, the operand stack, and the structure of the `.class` file in detail; makes it easier to read `javap -c` output.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 13th ed., Chapter 3: treats primitive types, conversions, and operators from an experienced programmer's point of view, pointing out common mistakes.

2.2.12 References

1. Oracle, *The Java Tutorials*: “Language Basics” (variables, operators, expressions, control flow), docs.oracle.com/javase/tutorial.
2. J. Gosling et al., *The Java Language Specification*, Java SE 21 Edition (Chapter 3: lexical structure, keywords, literals; Chapter 4: types; Chapter 5: conversions).
3. T. Lindholm et al., *The Java Virtual Machine Specification*, Java SE 21 Edition (bytecode and the JVM).
4. P. Deitel, H. Deitel, *Java: How to Program*, 11th ed., Pearson, 2017 (Chapters 2–5).
5. K. Sierra, B. Bates, *Head First Java*, 3rd ed., O’Reilly, 2022.
6. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 13th ed., 2024 (Chapter 3).
7. A. B. Downey, C. Mayfield, *Think Java*, 2nd ed., O’Reilly, 2019.

PART I · CHAPTER 3

Control Structures, Methods, Arrays and Strings

Decisions and loops steer the flow of a program. This chapter covers if and switch, nested loops, break and continue, and then methods: how to define and call them, scope rules and their role in abstraction. Arrays are examined through their memory layout, common mistakes, basic array algorithms and multidimensional arrays, reinforced with a Tic Tac Toe program. Strings are studied for immutability, comparison and formatted output, and recursion starts from factorial and series examples before moving on to efficiency, tail recursion and memoization.

NOTES IN THIS CHAPTER

3.1 Control Structures

3.2 Methods

3.3 Arrays

3.4 Strings

3.5 Recursion

PART I · CHAPTER 3.1

Control Structures

If a program consisted only of lines running from top to bottom, it would do the same work in the same order on every run. It could not behave differently depending on what the user enters, and doing a job a thousand times would take a thousand lines. Control structures remove both limitations. Decision structures look at a condition and choose one of two or more paths. Loops repeat a group of statements as long as a condition holds. First sequential flow is covered, then `if`, `if-else`, the `else if` chain, nested decisions, and `switch`. After that come the `for`, `while`, and `do-while` loops, nested loops, and `break` and `continue`, which alter the flow in the middle of a loop. For every structure the code is traced line by line, the values of the variables are shown in tables, and the Java compiler's real error messages are given. All the code was run with JDK 21, and the outputs are taken from those runs.

LEARNING OUTCOMES

Flow control
Explain how sequential, decision, and loop structures direct the flow of a program.

Decision structures
Write conditional code with `if`, `if-else`, the `else if` chain, nested decisions, and compound conditions using `&&` and `||`; predict the result of short-circuit evaluation.

switch-case
Compare a value against constants; explain the job of `break` and the fall-through behavior; tell the classic `switch` from the arrow-form `switch` expression.

Loops
Choose among the `for`, `while`, and `do-while` loops appropriately; avoid infinite loops and boundary errors.

Nested loops
Work out how many times nested loops run in pattern, table, and digit examples.

Tracing
Trace a program step by step to find the values of the variables and the screen output in advance; show the effect of `break` and `continue`.

3.1.1 Control structures

Java source code consists of statements. A variable declaration, an assignment, or a `println` call is a statement, and each ends with a semicolon. The compiler translates these statements to bytecode in the order they are written. At run time, it is the control structures that determine which statement runs, when, and how many times.

Definition 3.1.1

A **control structure** is a language construct that determines in what order and how many times statements run. The path the program follows according to this order is called its **flow**.

3.1.1.1 Three basic structures

Java has three basic control structures (Figure 3.1.1). In **sequential execution**, statements run from top to bottom, once each. In **decision structures** a condition is examined and one of two or more paths is chosen; Java's forms are `if-else` and `switch`. In **loops** the same group of statements is repeated as long as a condition holds; Java has the `for`, `while`, and `do-while` loops.

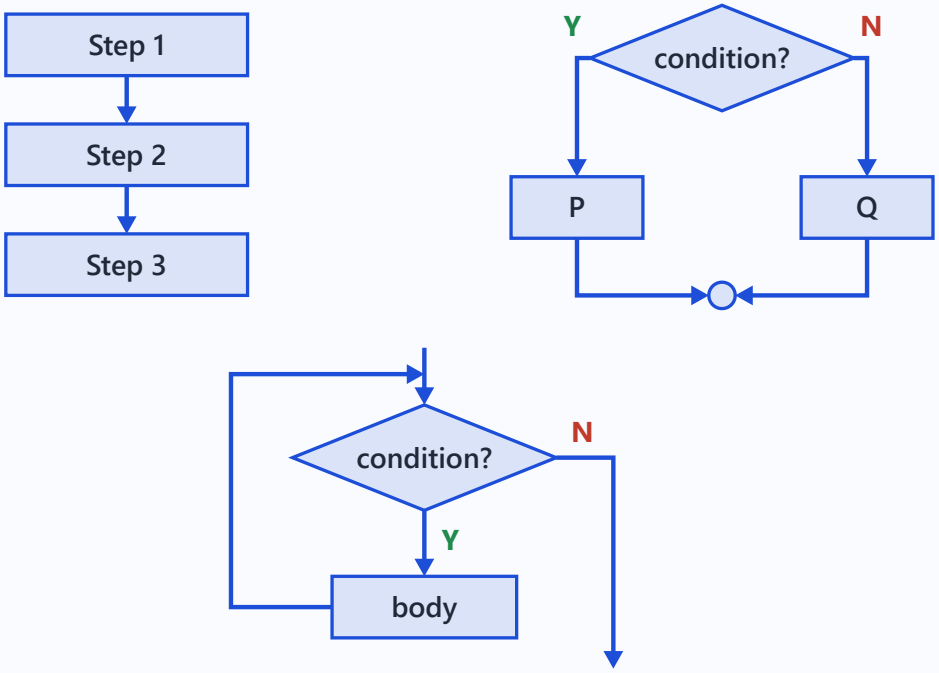


Figure 3.1.1: The three basic control structures. Left, sequential flow: the steps run one after another. Middle, a decision: if the condition is true (Y), P is done, if false (N), Q is done, and the two paths join again. Right, a loop: while the condition is true the body runs and the flow returns to the condition; when the condition is false the loop is left.

Small as they look, these three structures are enough to write every known algorithm. This result rests on the structured program theorem shown by Böhm and Jacopini in 1966: by combining sequence, selection, and repetition, every computable task can be expressed. The structures can also be nested. A loop body can contain a decision, and a branch of a decision can contain a loop. Real programs are the result of this nesting.

Decisions give a program flexibility. The same program answers different inputs differently: it turns a negative number positive, and it writes B for a student who scores 85 and F for one who scores 45. Loops, for their part, shorten repetitive work. To add the numbers from 1 to 1,000,000, a three-line loop is enough, not a million lines. The code gets shorter and easier to change: to raise the bound from 1,000,000 to 2,000,000, only one number changes.

3.1.1.2 Sequential execution

Every program without a control structure is sequential. The code runs line by line from top to bottom, and no line starts before the previous one has finished. The four lines below are the smallest example of this order:

```
int x = 5;
int y = 10;
int result = x + y;
System.out.println(result);
```

The first line sets aside an `int` variable named `x` and writes 5 into it. The second line does the same for `y` with 10. The third line reads the results of the previous two lines, computes 5 + 10 = 15, and writes it into the variable `result`. The last line prints this value on the screen. Table 3.1.1 shows which variables are in memory after each line. An empty cell means the variable has not yet been declared.

Line	Statement	x	y	result	Screen
1	`int x = 5;`	5			
2	`int y = 10;`	5	10		
3	`int result = x + y;`	5	10	15	
4	`System.out.println(result);`	5	10	15	15

Table 3.1.1: Line-by-line trace of the sequential program.



Figure 3.1.2: The three variables in memory after the third line. The 15 in the `result` box was computed

from the values written by the previous two lines.

The importance of order shows in an experiment. Move the third line to the very beginning: when the line `int result = x + y;` runs, `x` and `y` are not yet declared. In this case Java does not compile the program at all and gives the message `cannot find symbol`. In sequential structure each line depends on the state left by the lines before it.

Common mistake

Reading a variable without giving it a value. If `System.out.println(result);` is written right after the line `int result;`, the compiler reports the error `variable result might not have been initialized`. At compile time Java checks that a local variable certainly receives a value before it is read. This check matters most in decision structures that give the variable a value in only some of their branches.

3.1.2 Making decisions with if

Decision making means the program chooses at run time which path to follow. The choice cannot be made at compile time, because it often depends on input: the number the user types, the contents of a file that is read, a measurement from a sensor. Its counterpart in everyday life is a sign at a fork in the road: “If the weather is rainy, take an umbrella; if not, don’t.” In Java, decisions are written with `if-else` and `switch`. This section covers the `if` family, and the following sections cover multiway decisions and `switch`.

3.1.2.1 Conditions and comparison operators

Definition 3.1.2

A **condition** is a logical (boolean) expression whose result is `true` or `false`. In Java the type of a condition is always `boolean`.

Most conditions are built by comparing two values. Java’s six comparison operators are given in Table 3.1.2, with examples for `x = 7`.

Operator	Meaning	Example	Result
`==`	equal to	`x == 7`	`true`
`!=`	not equal to	`x != 7`	`false`
`<`	less than	`x < 10`	`true`
`>`	greater than	`x > 10`	`false`

Operator	Meaning	Example	Result
`<=`	less than or equal to	`x <= 7`	`true`
`>=`	greater than or equal to	`x >= 8`	`false`

Table 3.1.2: Comparison operators (`x = 7`).

No space goes between the characters of a two-character operator: if `< =` is written, the compiler reads it as two separate tokens and reports an error. The mathematical signs ≤, ≥, and ≠ have no counterpart in Java; `<=` , `>=` , and `!=` are used instead.

The most important distinction is between `=` and `==` . A single equals sign is **assignment**: it writes the value on the right into the variable on the left. A double equals sign is **comparison**: it asks whether two values are equal and produces a `boolean` . In C, `if (x = 7)` compiles and is always taken as true, which makes it a very insidious source of bugs there. Java, on the other hand, requires the condition to be `boolean` . Because the value of the expression `x = 7` is the `int` 7, the program does not compile:

```
error: incompatible types: int cannot be converted to boolean
        if (x = 7) {
              ^
```

This rule protects only numeric variables such as `int` . If the variable itself is `boolean` , `if (done = true)` compiles, writes `true` into `done` , and the condition always comes out true. The easiest way to avoid this trap is never to compare a `boolean` variable with anything: `if (done)` and `if (!done)` are enough.

3.1.2.2 The if-else structure

Definition 3.1.3

The **if statement** is a decision statement that runs a block if the condition is true. The optional `else` branch gives the block to run when the condition is false.

```
if (condition) {
    // runs if the condition is true
} else {
    // runs if the condition is false
}
```

The structure works in three steps. First the condition in parentheses is evaluated. If the condition is `true` , the `if` block runs and the `else` block is skipped. If the condition is `false` , the `if` block is skipped and the `else` block runs. In both cases the flow rejoins at the end of the structure and

continues with the next statement (Figure 3.1.3). At most one of the two blocks runs; if there is an `else`, exactly one runs.

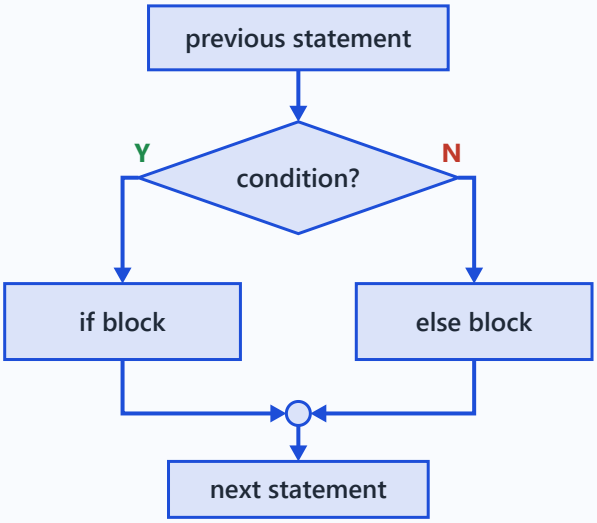


Figure 3.1.3: The `if-else` flow diagram. If the condition is true (Y) the `if` block runs, and if false (N) the `else` block runs; the two paths join and the flow moves on to the next statement.

There are two writing rules. The condition is always written in parentheses: `if condition` does not compile, `if (condition)` does. Blocks are opened and closed with braces. The group of statements between braces is called a **block**. A variable declared inside a block lives only in that block; when the block is left, its name is gone too.

3.1.2.3 if alone, and braces

The `else` branch is optional. If there is nothing special to do when the condition is false, only `if` is written:

```
int temp = 35;
if (temp > 30) {
    System.out.println("It's hot!");
}
System.out.println("The program continues.");
```

```
It's hot!
The program continues.
```

Because `temp` is 35, `35 > 30` is true and the block runs. Then the flow moves to the statement after the `if`. If the temperature were 20, the condition would be false, the block would be skipped, and only “The program continues.” would appear on the screen. A statement outside the `if` structure

runs in every case, independent of the result of the condition.

The braces may be left out when the block has a single statement. Java's rule is this: without braces, only the first statement is attached to the `if`. Indentation, that is, the spaces at the beginning of a line, has no meaning for Java. For someone used to languages like Python, which build blocks by indentation, this gives rise to the trap below:

```
int temp = 35;
if (temp > 40)
    System.out.println("Very hot!");
    System.out.println("Drink water.");
```

```
Drink water.
```

`35 > 40` is false, so "Very hot!" is not printed. The line "Drink water." is not part of the `if`, even though the two lines stand at the same indentation. The structure the compiler sees is: `if (temp > 40) println("Very hot!");` followed by a separate, unconditional `println("Drink water.");`. If the programmer's intention is to make both lines depend on the condition, only braces can do that. It is therefore a good habit to use `{ }` even for one-line blocks. When a second line is added to the block later, the same mistake does not arise again.

Common mistake

Putting a semicolon after the condition. `if (x > 5); { … }` compiles, but the semicolon by itself is an **empty statement**. The compiler reads this as "if the condition is true, do nothing." The block that follows is not attached to the `if` and runs in every case. Tried with `x = 3`, `if (x > 5); { System.out.println("x is greater than five"); }` still prints "x is greater than five." The compiler gives no warning for this line either.

3.1.2.4 An age check

The first complete decision example is a six-line piece of code that checks the age for a driver's license:

```
int age = 18;
if (age >= 18) {
    System.out.println("You can get a license.");
} else {
    System.out.println("You cannot get a license.");
}
```

The first line assigns 18 to the variable `age`. In the second line the condition is evaluated: `18 >= 18` is true, because `>=` includes equality. In the flow diagram this means going to the yes (Y) branch (Figure 3.1.4). The third line runs and "You can get a license." is printed. The `else` block, that is, the fifth line, is skipped.

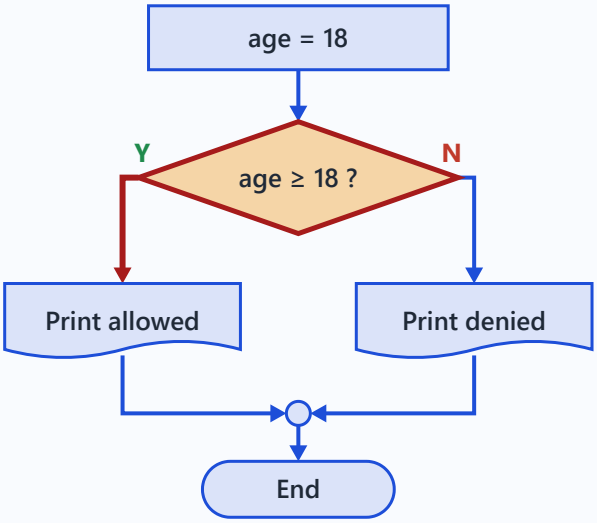


Figure 3.1.4: The flow diagram of the age check. For `age = 18` the condition is true and the flow goes to the Y branch, drawn in bold.

`age`	`age >= 18`	Block that runs	Screen
16	`false`	`else`	You cannot get a license.
18	`true`	`if`	You can get a license.
25	`true`	`if`	You can get a license.

Table 3.1.3: The result of the age check for three values.

The 18 in the table is a **boundary value**: the point where the condition changes from true to false. Reliable code should be tried on both sides of the boundary and exactly on it. If the condition had mistakenly been written `age > 18`, 16 and 25 would give the right result and only 18 the wrong one. A mistake of this kind shows up only in a test with the boundary value.

3.1.2.5 Absolute value

The second example reads a floating-point number from the user and prints its absolute value. The absolute value of a number is the number with its sign changed if it is negative, and the number itself otherwise: |−7.5| = 7.5 and |3| = 3.

```
Scanner scanner = new Scanner(System.in);
System.out.print("Enter a number: ");
double num = scanner.nextDouble();
double absValue;
if (num < 0) {
    absValue = -num;
} else {
    absValue = num;
}
System.out.println("Absolute value: " + absValue);
```

The code assumes that the line `import java.util.Scanner;` is at the top of the file. The first line creates a `Scanner` object to read from the keyboard. The second line tells the user what to enter. Here `print` is used, not `println`: `print` does not move to a new line, and the cursor waits right next to the question. In the third line `nextDouble()` reads the number the user typed.

The fourth line declares the variable `absValue` but gives it no value. The `if-else` below will give it one. Java accepts this line because each of the two branches writes a value into the variable. The compiler can check this: whichever branch runs, by the time the last line is reached the variable has certainly received a value. If the `else` branch were deleted, the compiler would report `variable absValue might not have been initialized` for the last line.

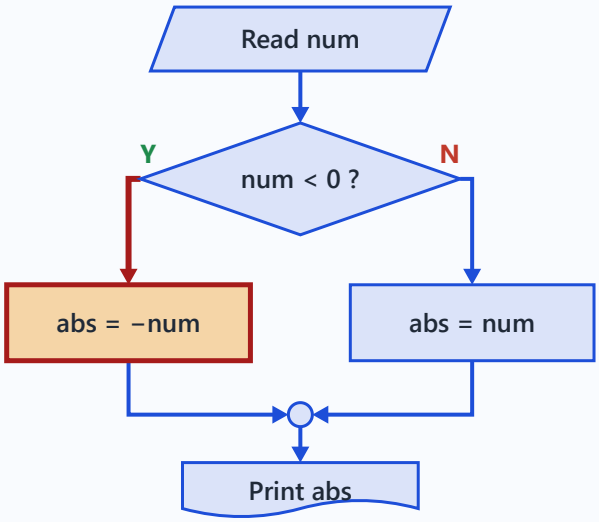


Figure 3.1.5: The absolute-value flow diagram. When −7.5 is entered, `num < 0` is true; in the highlighted box `absValue = -num` runs.

Suppose the user enters −7.5. On the fifth line `-7.5 < 0` is true. On the sixth line the expression `-num` computes −(−7.5) = 7.5 and writes it into `absValue`. The `else` block is skipped. The last line prints `Absolute value: 7.5`. Table 3.1.4 summarizes running the same program with four inputs.

From the keyboard	`num`	`num < 0`	Screen
-7,5	−7.5	`true`	Absolute value: 7.5
7,5	7.5	`false`	Absolute value: 7.5
0	0.0	`false`	Absolute value: 0.0
-7.5	—	—	`InputMismatchException`

Table 3.1.4: The absolute-value program on a computer with a Turkish locale.

The last row is a detail specific to Java. `Scanner` reads numbers according to the computer's regional setting. In the Turkish setting the decimal separator is a comma. That is why `-7,5` is accepted, while `-7.5` is not taken as a number and the program stops with this message:

```
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:947)
        at java.base/java.util.Scanner.next(Scanner.java:1602)
        at java.base/java.util.Scanner.nextDouble(Scanner.java:2573)
```

Reading follows the regional setting; writing does not. `println` always writes a `double` value with a period: `7.5`. This is why the same program shows a comma on input and a period on output. If reading is to use a period as well, `scanner.useLocale(java.util.Locale.US)` is called after the `Scanner` is created.

3.1.3 Multiway decisions

An `if-else` structure chooses between two paths. In real problems there are often more options. A number can be positive, negative, or zero; a grade can correspond to A, B, C, D, or F. Java offers two tools for these cases: the `else if` chain, which strings conditions one after another, and nested structure, which places decisions inside one another. At the end of the section come the `&&` and `||` operators, which combine several conditions in one expression, and the conditional operator `?:`.

3.1.3.1 The else-if chain

Definition 3.1.4

An **else-if chain** is a structure that tries the conditions in order from top to bottom and runs the block of the first true condition. If no condition is true, the final `else` block, if there is one, runs.

The complete program below finds the sign of a number. The three possible outcomes are separated by two conditions and a final `else`:

```java
import java.util.Scanner;

public class PositiveNegative {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number: ");
        double num = scanner.nextDouble();
        if (num > 0) {
            System.out.println("Positive.");
        } else if (num < 0) {
            System.out.println("Negative.");
        } else {
            System.out.println("Zero.");
        }
    }
}
```

When the program is run with three different inputs, it gives these outputs:

```
Enter a number: 4
Positive.
```

```
Enter a number: -2
Negative.
```

```
Enter a number: 0
Zero.
```

When 4 is entered, the first condition `4 > 0` is true. “Positive.” is printed and the rest of the chain is never looked at. When −2 is entered, the first condition is false and the flow moves to the second condition: `-2 < 0` is true and “Negative.” is printed. When 0 is entered, both conditions are false and the last `else` runs. None of the three branches can run at the same time: the moment one branch is selected, the chain ends.

`else if` is not a separate keyword in Java. The structure is just a new `if` written inside an `else` branch. The form of the program in the compiler’s eyes is: `if (num > 0) { … } else { if (num < 0) { … } else { … } }`. Programmers write it at a single level, one below another, because that is more readable. Each link in the chain is evaluated knowing that all the conditions before it were false. Only non-positive numbers reach the condition `num < 0` in the second link.

3.1.3.2 Tracing the letter-grade chain

The letter-grade example shows best how the chain uses order. A numeric grade of 90 or above gets A, 80–89 gets B, 70–79 gets C, 60–69 gets D, and below 60 gets F.

```
int grade = 75;
if (grade >= 90) {
    System.out.println("A");
} else if (grade >= 80) {
    System.out.println("B");
} else if (grade >= 70) {
    System.out.println("C");
} else if (grade >= 60) {
    System.out.println("D");
} else {
    System.out.println("F");
}
```

For `grade = 75` the flow is as follows. The first condition `75 >= 90` is false and the flow moves to the next `else if`. `75 >= 80` is false too. `75 >= 70` is true: the first true condition has been found. “C” is printed. The remaining condition, `grade >= 60`, is never tested (Figure 3.1.6).

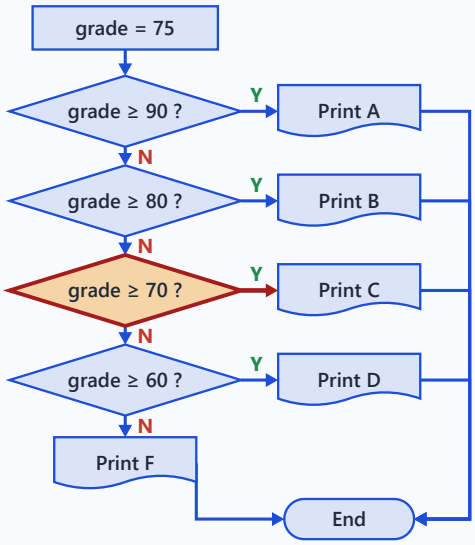


Figure 3.1.6: The flow diagram of the `else-if` chain, `grade = 75`. At the first two diamonds the flow goes to the N branch; at the third diamond (`grade ≥ 70`) the condition is true and the flow moves to the “Print C” box.

`grade`	`>= 90`	`>= 80`	`>= 70`	`>= 60`	Conditions tested	Screen
95	true	—	—	—	1	A
80	false	true	—	—	2	B
75	false	false	true	—	3	C

`grade`	`>= 90`	`>= 80`	`>= 70`	`>= 60`	**Conditions tested**	**Screen**
60	false	false	false	true	4	D
45	false	false	false	false	4	F

Table 3.1.5: Conditions tested in the letter-grade chain.

Notice that upper bounds such as `grade < 90` are not written in the conditions. The link `else if (grade >= 80)` is reached only if `grade >= 90` is false; so at this point the grade is already below 90. The chain takes its upper bound from the order itself. That is why the order of the conditions matters. Let the same four conditions be written in reverse order:

```
if (grade >= 60) {
    letter = "D";
} else if (grade >= 70) {
    letter = "C";
} else if (grade >= 90) {
    letter = "A";
} else {
    letter = "F";
}
```

This code produces “D” for 95 and for 75 alike. Every number from 60 up satisfies the first condition, and the chain ends at the first link. No value can reach the `grade >= 90` link. The compiler does not catch this mistake, because the code follows the rules of Java. What is wrong is the logic. The rule is simple: in a chain written with `>=`, the thresholds go from largest to smallest; in a chain written with `<`, from smallest to largest.

Common mistake

Writing independent `if` statements one after another instead of a chain. In code written as `if (grade >= 90) letter = "A"; if (grade >= 80) letter = "B"; if (grade >= 70) letter = "C";`, each `if` is tested separately. For 95, first `"A"`, then `"B"`, then `"C"` is assigned, and the result is C. The word `else` carries the meaning “if the previous conditions were false”; without it the conditions do not exclude one another.

3.1.3.3 Nested decisions

Definition 3.1.5

A **nested decision** is a decision structure written inside the block of another decision structure. The inner decision is tested only when the outer condition selects the branch that contains it.

```
if (cond1) {
    if (cond2) {
        // cond1 AND cond2 true
    }
} else {
    // cond1 false
}
```

Nested structure is useful when a second question is to be asked depending on the answer to the first (Figure 3.1.7). The body of the inner `if` is entered only if `cond1` and `cond2` are both true. If `cond1` is false, the inner `if` is never tested and the outer `else` runs directly.

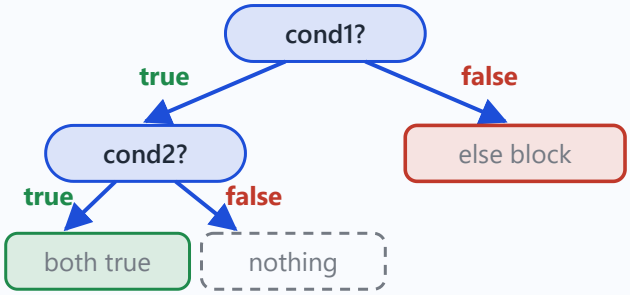


Figure 3.1.7: A nested decision tree. If `cond1` is true, `cond2` is tested; if false, the outer `else` block runs. If `cond1` is true and `cond2` is false, no block runs.

A two-level version of the letter-grade example makes the structure concrete. First it checks whether the student passes, and for those who pass it separates A from B:

```
int grade = 85;
if (grade >= 60) {
    if (grade >= 90) {
        System.out.println("Letter grade: A");
    } else {
        System.out.println("Letter grade: B");
    }
} else {
    System.out.println("Letter grade: F");
}
```

For `grade = 85` the outer condition `85 >= 60` is true and the outer `if` block is entered. There the inner condition is tested: `85 >= 90` is false, the flow goes to the inner `else`, and “Letter grade: B” is printed. The outer `else` is skipped (Figure 3.1.8). With `grade = 45`, the outer condition would be false, the inner `if` would never be tested, and “Letter grade: F” would be printed directly.

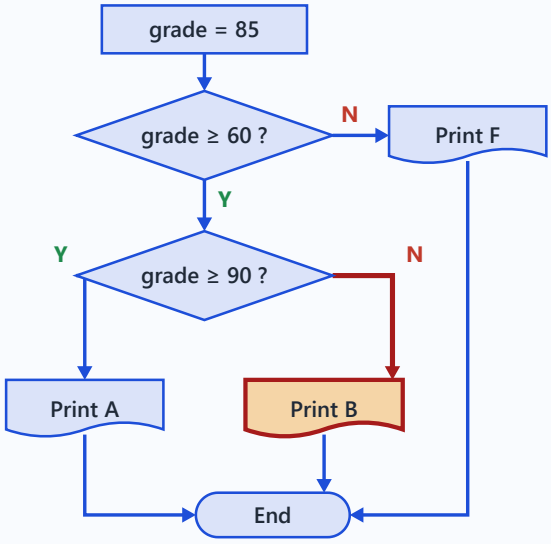


Figure 3.1.8: The flow diagram of the nested decision, `grade = 85`. At the outer diamond the flow goes to the Y branch, at the inner diamond to the N branch, and the "Print B" box runs.

Very deeply nested structures lead to code that is hard to read. In an `if` nested four or five levels deep, you have to count braces to find which `else` belongs to which `if`. There are two remedies. The first is proper indentation: each inner level starts four spaces further in. The second is to flatten the structure. Code that cares only about the case where two conditions are both true can be reduced to a single `if` with `&&`: `if (age >= 18) { if (hasLicense) … }` and `if (age >= 18 && hasLicense) …` do the same job.

The dangling else problem. In nested `if` statements without braces, it can seem unclear which `if` an `else` belongs to. Java's rule is exact: an `else` attaches to the **nearest** preceding `if` that does not yet have an `else`. In the code below the indentation makes the `else` look attached to the inner `if`, and the rule gives the same result:

```
int x = 7;
if (x > 5)
    if (x > 10)
        System.out.println("A");
    else
        System.out.println("B");
```

For `x = 7`, `7 > 5` is true, the inner condition `7 > 10` is false, and the `else` of the inner `if` prints "B". When the same code is run with `x = 3`, it prints nothing: the outer condition is false and the outer `if` has no `else`. Even if the `else` is shifted left to the indentation of the outer `if`, the result does not change; indentation means nothing to the compiler. If the `else` is really meant to attach to the outer `if`, the inner `if` is closed with braces:

```
int x = 7;
if (x > 5) {
    if (x > 10) {
        System.out.println("A");
    }
} else {
    System.out.println("B");
}
```

This version prints nothing for `x = 7` and “B” for `x = 3`. The two pieces of code consist of the same lines, but the braces have changed the owner of the `else`.

3.1.3.4 Compound conditions and short-circuiting

Several conditions are combined into a single condition with logical operators. Java has three logical operators: `&&` (and), `||` (or), and `!` (not). `a && b` is true only if both sides are true. `a || b` is true if at least one of the sides is true. `!a` is the opposite of `a`.

a	b	a && b	a || b	!a
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Table 3.1.6: Truth table of the logical operators.

The most common use of logical operators is range checking. Whether a grade lies between 60 and 89 cannot be written as in mathematics, `60 <= grade < 90`. Java reads this expression from left to right: first `60 <= grade` is computed and a `boolean` results. Then this `boolean` is compared with 90, but one cannot ask whether a `boolean` is less than 90. The compiler gives this message:

```
error: bad operand types for binary operator '<'
  first type:  boolean
  second type: int
```

The correct form joins the two comparisons with `&&`: `grade >= 60 && grade < 90`. Using `||` here is also a common mistake. `grade >= 60 || grade < 90` is true for every number: every number below 60 is also below 90.

Definition 3.1.6

Short-circuit evaluation: `&&` and `||` evaluate the left side first. If the result is already determined by the left side alone, the right side is not evaluated at all. In `&&`, if the left side is `false` the result is `false`; in `||`, if the left side is `true` the result is `true`.

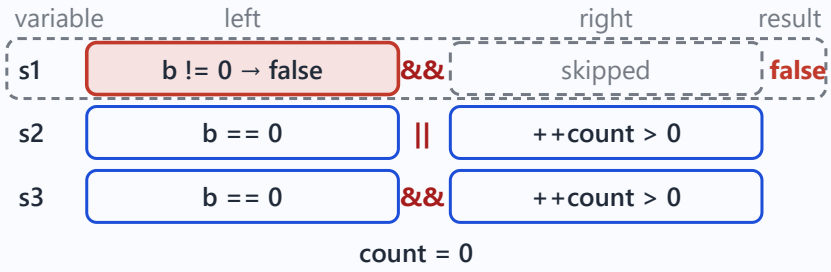
Short-circuiting is more than a detail of speed; it changes the behavior of the program. The code below shows this in three lines:

```
int a = 10, b = 0, count = 0;
boolean s1 = b != 0 && a / b > 2;
boolean s2 = b == 0 || ++count > 0;
boolean s3 = b == 0 && ++count > 0;
System.out.println(s1 + " " + s2 + " " + s3);
System.out.println("count = " + count);
```

```
false true true
count = 1
```

At the start `b = 0`. If `a / b` were evaluated, a division-by-zero error (`ArithmeticException`) would occur. `count` is there to count whether the right-hand sides run: each time `++count` runs, it increases by 1.

- `s1`: `&&` evaluates the left side first: `b != 0` is false. If the left side is false, the result of `&&` is false whatever the right side is. `a / b > 2` is never evaluated and no error occurs. `s1 = false`.
- `s2`: for `||` the left side `b == 0` is true. The result is already true; `++count` does not run and `count` stays 0. `s2 = true`.
- `s3`: `&&` again; the left side `b == 0` is true. The result is not yet determined, so the right side must be evaluated as well. `++count` runs and `count` becomes 1; `1 > 0` is true. `s3 = true`.



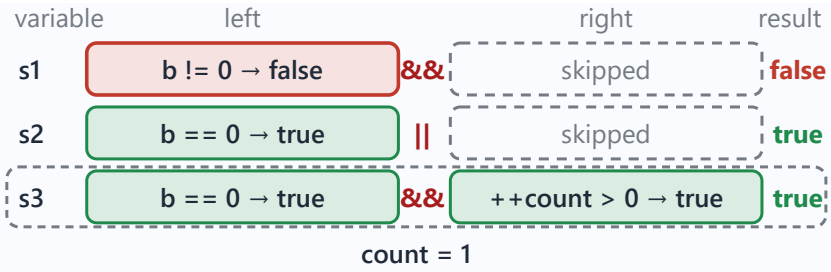


Figure 3.1.9: Two moments of short-circuit evaluation. Top: `s1` has been computed; because the left side is `false`, the right side is marked "skipped." Bottom: all three lines are finished; only the right side of `s3` ran and `count` became 1.

In the end only one of the three right-hand sides ran. This behavior has two sides. The useful side is guarding a dangerous operation, as in `s1`. When `b != 0 && a / b > 2` is written, the division is performed only if the denominator is nonzero. The same pattern is common with objects: `s != null && s.length() > 0`. The dangerous side is side effects. If an operation that changes a variable, such as `++count`, is written on the right-hand side, whether it runs depends on the left side. A person reading the code easily misses this. The safest course is not to write operations with side effects to the right of `&&` and `||`.

Java has two more operators that do not short-circuit: `&` and `|`. When used with `boolean` values they always evaluate both sides. If `b != 0 & a / b > 2` is written, the division is performed even though the left side is false, and the program stops with `java.lang.ArithmeticException: / by zero`. In conditions, writing a single `&` is most often a typing mistake.

3.1.3.5 The conditional operator

Choosing one of two values according to a condition is a very common task. For it an `if-else` can be written:

```
int a = 17, b = 42;
int max;
if (a > b) {
    max = a;
} else {
    max = b;
}
System.out.println(max);
```

Java offers the **conditional operator**, which does the same job in a single expression. Because it has three parts, it is also called the ternary operator: `condition ? value1 : value2`. If the condition is true, the value of the expression is `value1`; if false, `value2`.

```
int a = 17, b = 42;
int max = (a > b) ? a : b;
System.out.println(max);
```

Because `17 > 42` is false, the second value is chosen and 42 is printed. `if-else` is a **statement** and does a job. `?:` is an **expression** and produces a value. It can therefore be written on the right of an assignment, as the argument of a method, or inside a string concatenation: `System.out.println(grade >= 60 ? "passed" : "failed");`. Nested conditional operators (`a ? b : c ? d : e`) quickly become unreadable; with more than two alternatives an `else if` chain is preferred. To find the larger of two numbers, Java's ready-made `Math.max(a, b)` method can also be used.

3.1.4 switch-case

Some decisions always ask whether the same variable equals different constant values: is the month 1, 2, 3...? Is the operation `'+'` or `'-'`? Such decisions can also be written with an `else if` chain, but in a twelve-link chain `month ==` is repeated in every link. The `switch` structure is designed for this situation.

3.1.4.1 Structure and operation

Definition 3.1.7

The **switch statement** compares the value of the expression in parentheses with the constants in the `case` labels and continues with the code under the matching label. If no label matches, the code under the `default` label, if there is one, runs.

```
switch (variable) {  // value being checked
    case value1:     // one value per case
        // statements 1
        break;
    case value2:
        // statements 2
        break;
    // ...
    default:         // if nothing matches
        // default statements (optional)
}
```

The expression in parentheses is called the **selector**. The selector is evaluated once. Its value is compared with the `case` labels from top to bottom. When a matching label is found, the statements under it run. On reaching `break`, the `switch` is left and the flow moves to the statement after the closing brace. If no label matches, the `default` section runs. If there is no `default` either, the `switch` is left without anything being done (Figure 3.1.10).

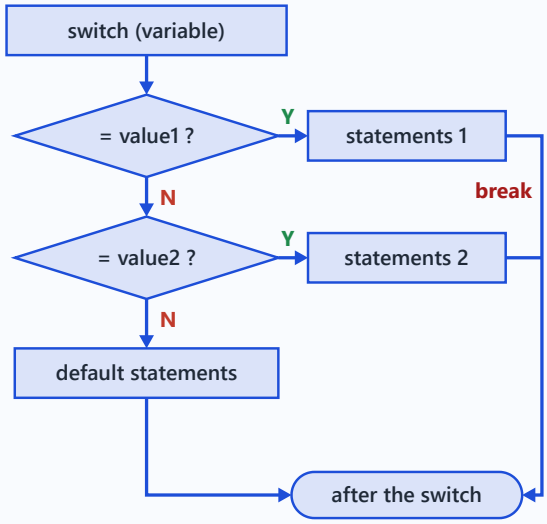


Figure 3.1.10: How `switch-case` works. The selector is compared in turn with `value1` and `value2`. The statements of the matching `case` run and the `switch` is left with `break`; if none matches, the `default` statements run.

An everyday analogy is the buttons in an elevator. The floor number pressed selects the floor to go to directly. The elevator does not stop at every floor to ask "is this the 3rd floor?" A `switch` often works that way too: conceptually the labels can be thought of as tried in turn, but if the labels are closely packed integers, the compiler builds a jump table and jumps directly to the right label.

3.1.4.2 Determining the season

The code that finds the season from the month number shows all the parts of `switch` together:

```
int month = 4;
String season;
switch (month) {
    case 3: case 4: case 5:
        season = "Spring";
        break;
    case 6: case 7: case 8:
        season = "Summer";
        break;
    case 9: case 10: case 11:
        season = "Autumn";
        break;
    case 12: case 1: case 2:
        season = "Winter";
        break;
    default:
        season = "Unknown";
}
System.out.println(month + ": " + season);
```

```
4: Spring
```

The first line assigns 4 to `month`. The second line declares `season` but gives it no value. On the third line the selector `month` is evaluated once: its value is 4. Then the matching label is looked for. The fourth line has three labels written on one line: `case 3: case 4: case 5:`. 4 matches one of them. There is a single piece of code under the three labels; whichever matches, the flow drops into the same code. `"Spring"` is assigned to `season`. `break` carries the flow to just past the closing brace of the `switch`. The last line prints "4: Spring."

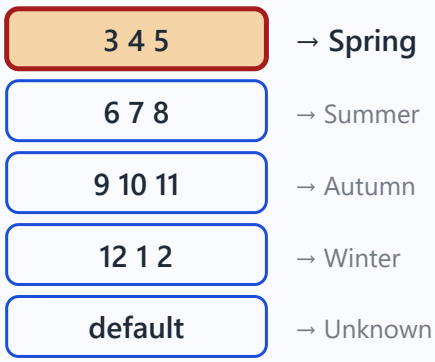


Figure 3.1.11: The `case` groups in the season example. `month = 4` matches the first group and `season` becomes "Spring."

The `default` branch does two jobs here. The first is to give an answer for invalid month numbers such as 0, 13, or −5. The second is to let the code pass the compiler's variable check. If `season` were assigned only in the `case` branches, the compiler would reason as follows: if the selector matches no label, `season` never gets a value. The read on the last line is therefore rejected:

```
error: variable season might not have been initialized
        System.out.println(season);
                           ^
```

Once `default` is added, every path out of the `switch` has written a value into `season` and the error disappears. When all month values from 0 to 13 are tried, "Unknown" is printed for 0 and 13 and the correct season for the others.

3.1.4.3 Fall-through

Definition 3.1.8

Fall-through is the running of the code of the `case` labels below as well, after the code of a `case` that does not end with `break`.

A `case` label is not a comparison statement but an **entry point**. The `switch` only chooses where to start. Once the flow has entered, it does not look at the labels below; it moves line by line until a `break` or the end of the `switch`. In the code below, the first two `case` labels have no `break`:

```
int day = 2;
switch (day) {
    case 1:
        System.out.println("Monday");
    case 2:
        System.out.println("Tuesday");
    case 3:
        System.out.println("Wednesday");
        break;
    default:
        System.out.println("Other day");
}
System.out.println("switch done");
```

```
Tuesday
Wednesday
switch done
```

With `day = 2` the selector is evaluated. `case 1` is skipped, `case 2` matches, and “Tuesday” is printed. There is no `break` here. The flow arrives at the line `case 3:` below. This label does not ask “is the day 3?”; the flow falls into the code under it and “Wednesday” is printed too, although the day is not 3. The `break` that follows leaves the `switch`, and `default` does not run. Finally “switch done” is printed (Figure 3.1.12).

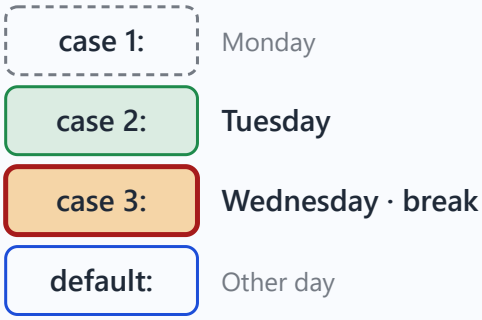


Figure 3.1.12: Fall-through. `case 2` ran (green), and because there is no `break` the flow falls into `case 3` (highlighted). `case 1` was skipped.

`day`	Entry point	Code that runs	Screen
1	`case 1`	case 1, case 2, case 3, `break`	Monday, Tuesday, Wednesday
2	`case 2`	case 2, case 3, `break`	Tuesday, Wednesday
5	`default`	default	Other day

Table 3.1.7: The output of the same `switch` for three values.

Fall-through is not always a mistake. The line `case 3: case 4: case 5:` in the season example is in fact a deliberate use of this behavior: `case 3` and `case 4` are empty labels and the flow falls into the common code below. The mistake is forgetting `break` where the fall-through is not wanted. The compiler gives no warning in that case. Deliberately left fall-throughs are therefore usually marked with a comment: `// no break: fall-through intended`.

Example 3.1.1

What does the following code print?

```
int k = 2;
int points = 0;
switch (k) {
    case 1:  points += 10;
    case 2:  points += 20;
    case 3:  points += 30;  break;
    case 4:  points += 40;
}
System.out.println(points);
```

The selector is 2; the flow enters at `case 2` and `points` becomes 20. Because there is no `break`, it falls into `case 3` and `points` becomes 50. The `break` on this line leaves the `switch`; `case 4` does not run. 50 is printed. With `k = 1` it would print 10 + 20 + 30 = 60, and with `k = 4` it would print 40.

3.1.4.4 switch with char and String

The selector does not have to be an `int`. A simple calculator can be written with a `char` selector:

```
int a = 12, b = 4;
char op = '*';
int result;
switch (op) {
    case '+':  result = a + b;  break;
    case '-':  result = a - b;  break;
    case '*':  result = a * b;  break;
    case '/':  result = a / b;  break;
    default:
        System.out.println("Bad operator");
        result = 0;
}
System.out.println(a + " " + op + " " + b
        + " = " + result);
```

```
12 * 4 = 48
```

`char` literals are written in single quotes: `'+'`, `'*'`. Short `case` branches can be fitted on one line: `case '+': result = a + b; break;`. Line breaks do not matter to the compiler. For `op = '*'` the third label matches, `result` becomes 48, and “12 * 4 = 48” is printed. For an undefined operation such as `op = '%'`, `default` runs: “Bad operator” is printed, followed by “12 % 4 = 0.” The line `result = 0;` in `default` is required here as well; without it the last line gives a `might not have been initialized` error.

There is a danger in this code that is easy to overlook: for `b = 0` and `op = '/'`, `a / b` runs and the program stops with `ArithmeticException: / by zero`. In a real calculator the `'/'` branch would first have to check the condition `b != 0`.

Since Java 7 the selector can also be a `String`:

```
String command = "stop";
switch (command) {
    case "start":
        System.out.println("Engine started");
        break;
    case "stop":
        System.out.println("Engine stopped");
        break;
    default:
        System.out.println("Unknown command");
}
```

```
Engine stopped
```

The labels are string literals in double quotes. The comparison is made on content, as the `equals` method does; it is not the object-identity comparison done by `==`. The comparison is case-sensitive: for `command = "Stop"` no label matches and “Unknown command” is printed. If the text the user types is converted to lowercase first, the problem goes away:

```
String command = "Stop";
switch (command.toLowerCase()) {
    case "start":
        System.out.println("Engine started");
        break;
    case "stop":
        System.out.println("Engine stopped");
        break;
    default:
        System.out.println("Unknown command");
}
```

This code prints “Engine stopped.” With Turkish text, `toLowerCase()` depends on the computer’s regional setting: in the Turkish setting the letter “I” becomes “ı.” For commands made only of ASCII letters, using `toLowerCase(java.util.Locale.ROOT)` removes this difference.

3.1.4.5 switch rules

The rules of the classic `switch` exist so that the structure can be turned into a jump table. Each rule is checked with a compiler message (Table 3.1.8).

- **The type of the selector.** It can be `byte`, `short`, `char`, `int`, their wrappers (`Byte`, `Short`, `Character`, `Integer`), `String`, and `enum`. A classic `switch` cannot be written with `long`, `float`, `double`, or `boolean`.
- **Labels are constants.** `case` values must be constants known at compile time: `5`, `'a'`, `"stop"`, or a `final` variable with a constant value. An ordinary variable cannot be a label.
- **Labels are unique.** The same value cannot be used in two `case` labels.
- **Only equality is tested.** A range (`month >= 3 && month <= 5`) or a compound condition cannot be written in a `case`. `if-else` is needed for those.

Code	Compiler message
`double d = 2.5; switch (d) { … }`	`error: selector type double is not allowed`
`long d = 2; switch (d) { … }`	`error: selector type long is not allowed`
`int b = 2; switch (a) { case b: break; }`	`error: constant expression required`
`switch (a) { case 1: break; case 1: break; }`	`error: duplicate case label`

Table 3.1.8: Violations of the `switch` rules and the JDK 21 compiler's messages.

If the `b` in the third row were declared as `final int b = 2;`, the code would compile. The value of a `final` variable cannot change, and if it is given by a constant expression the compiler uses it like a constant. Why there is such a rule becomes clear when you think of the labels being written into a table at compile time: the compiler building the table cannot know a value that might change at run time.

3.1.4.6 switch or else-if?

The two structures do the same job with different strengths. Table 3.1.9 brings the differences together.

	`if-else` / `else if`	`switch-case`
What is tested?	Any `boolean` condition: ranges, compound conditions, different variables	Equality of a single expression with constant values
Flexibility	More flexible	Only certain types and constant `case` values
With many alternatives	Long chains can look complicated	A neat, readable list
Trap	If the order of conditions is wrong, the wrong branch is chosen	If `break` is forgotten, execution falls into the `case` labels below
Example	`if (grade >= 90) … else if (grade >= 80) …`	`switch (month) { case 3: … }`

Table 3.1.9: Comparison of the `if-else` chain and `switch-case`.

Every `switch` can be written as an `else if` chain: each `case` becomes an `==` comparison and `default` becomes the last `else`. The reverse is not always possible. Ranges such as `grade >= 90` in the letter-grade example can be carried over to `switch` only by writing eleven separate labels from 90 to 100. The rule of choice can be summed up as follows: use `switch` if a single variable is compared with a few constants, and `if-else` if there are ranges or several variables.

3.1.4.7 The switch expression

Java 14 added a new form to `switch`. The season example is written in this form as follows:

```
int month = 11;
String season = switch (month) {
    case 3, 4, 5 -> "Spring";
    case 6, 7, 8 -> "Summer";
    case 9, 10, 11 -> "Autumn";
    case 12, 1, 2 -> "Winter";
    default -> "Unknown";
};
System.out.println(season);
```

```
Autumn
```

Definition 3.1.9

A **switch expression** is the form of `switch` that uses arrow (`->`) labels and produces a value. It has been a standard part of the language since Java 14.

The new form has four differences. First, a `case` can take several values separated by commas: `case 3, 4, 5`. Second, there is no fall-through in the arrow form. The expression or block on the right of each branch runs and the `switch` ends; `break` is not needed. Third, `switch` produces a value, and this value can be assigned directly to a variable. Because the assignment is a statement, a semicolon follows the closing brace: `};`. Fourth, the expression form must cover all possibilities. When the selector is an `int`, the possible values are practically endless, so `default` is required. If `default` is deleted, the compiler gives this message:

```
error: the switch expression does not cover all possible input values
```

If a branch needs several statements instead of a single value, the right-hand side becomes a block and the value to be produced is given with `yield`:

```
int day = 3;
String type = switch (day) {
    case 6, 7 -> "weekend";
    default -> {
        System.out.println("counting working days");
        yield "weekday";
    }
};
System.out.println(type);
```

```
counting working days
weekday
```

The arrow form and the colon form cannot be mixed in the same `switch`; the compiler gives the message `different case kinds used in the switch`. The arrow form can also be used as a statement that produces no value: `switch (day) { case 6, 7 -> System.out.println("holiday"); default -> System.out.println("work"); }`. In this course the classic form is the basis, because most old code is written with it and the fall-through behavior is seen only in it. In new code written for Java 14 and later, the arrow form is safer: the mistake of forgetting `break` cannot be made in this form.

3.1.5 Application: temperature conversion

Before moving from decisions to loops, a small sequential program shows how formulas are translated to Java. The problem is to convert a Celsius value to the Fahrenheit and Kelvin scales. A few common points of the three scales are given in Table 3.1.10.

Celsius (°C)	Fahrenheit (°F)	Kelvin (K)	Meaning
−40	−40	233.15	The point where the two scales meet
0	32	273.15	Freezing point of water
25	77	298.15	Room temperature
100	212	373.15	Boiling point of water (at sea level)

Table 3.1.10: Corresponding values on the temperature scales.

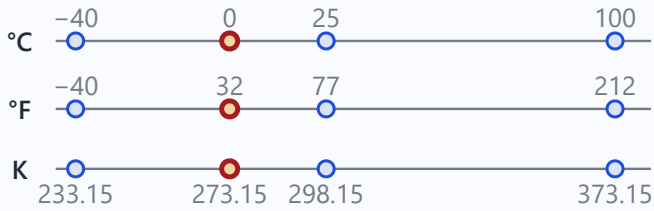


Figure 3.1.13: The three temperature scales on one axis. The red dots are the counterparts of 0 °C, the freezing point of water, on the three scales.

The conversion formulas are: °F = (°C × 1.8) + 32, °C = (°F − 32) / 1.8, and K = °C + 273.15. The Java program computes the first and last formulas for a single value:

```
public class Program {
    public static void main(String[] args) {
        double celsius = 25.0;
        double fahrenheit = (celsius * 1.8) + 32;
        double kelvin = celsius + 273.15;
        System.out.println(celsius + " °C = " + fahrenheit + " °F = " + kelvin + " K");
    }
}
```

```
25.0 °C = 77.0 °F = 298.15 K
```

The program is sequential: three lines of computation and one line of printing; there is no decision or loop. `double` is used for floating-point values. The `1.8` and `273.15` in the code are Java literals, and in code the decimal separator is always the period. The parentheses show the precedence explicitly: first `celsius * 1.8`, then `+ 32`. Because multiplication is done before addition, the result would be the same without parentheses, but they make the formula easier to read. The `+` signs in the printing line are not addition but concatenation: a `+` next to a string (`String`) converts the number to text and appends it. `double` values are written in the form `25.0`.

Floating-point numbers cannot hold every value exactly. Because 1.8 cannot be written as a finite fraction in binary, some results carry a small error. For 37 °C the expression `celsius * 1.8 + 32` prints `98.60000000000001`; for −40 °C, `celsius + 273.15` prints `233.14999999999998`. These numbers are not wrong; they only carry a rounding trace in the last digit. To make them look right on screen, they are printed rounded to two digits with `System.out.printf("%.2f%n", kelvin)`. Unlike `println`, `printf` follows the regional setting: on a computer with the Turkish setting the output is `233,15`.

Common mistake

Writing the formula with integers. For `int c = 37;` the expression `9 / 5 * c + 32` gives 69: `9 / 5` is the quotient of two `int` values, the fraction is discarded, and the result is 1. `c / 5 * 9 + 32` gives 95 because of 37 / 5 = 7. `c * 9 / 5 + 32` is computed from left to right: 333 / 5 = 66, result 98; still 98, not 98.6. For a fractional result at least one of the operands must be `double`: `c * 1.8 + 32` or `c * 9.0 / 5 + 32`.

3.1.6 Loops and for

Definition 3.1.10

A **loop** is a control structure that repeats a group of statements as long as a condition holds. The repeated group of statements is called the **body**, and each run of the body is called an **iteration**.

The everyday counterpart of a loop is running ten laps on a track. The same movements are made on each lap and the lap counter goes up by one. When the counter reaches ten, the run ends. It is the same in a program: the body stays the same, one or a few variables change on each iteration, and a condition says when to stop. Loops are also used to process a collection of data in order. Looking at every element of an array, reading every line of a file, or averaging a series of measurements is done with a loop. Instead of writing the same line 100 times, a single loop is written; the code gets shorter and the number of repetitions is set by changing a single number.

3.1.6.1 Kinds of loops

Java has three loops. All three rest on the same idea; the difference is where the condition is tested and where the initialization and update are written.

- `for`: used when the number of iterations is known. The initialization, condition, and update are on a single line: `for (int i = 0; i < 5; i++) { … }`.
- `while`: runs as long as a condition holds; the number of iterations may not be known in advance: `while (condition) { … }`.
- `do-while`: the condition is written at the end; the body runs at least once: `do { … } while (condition);`.

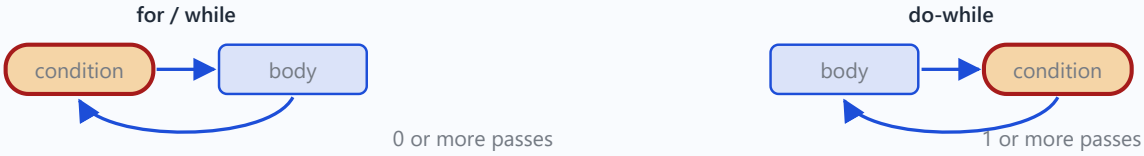


Figure 3.1.14: Where the condition goes. `for` and `while` test the condition before the body; the body may never run (at least 0 iterations). `do-while` tests the condition after the body; the body runs at least once.

3.1.6.2 The for loop

Definition 3.1.11

A **for loop** is a loop that gathers the initialization, condition, and update steps in its header on a single line, separated by semicolons.



Figure 3.1.15: The parts of the `for` header. ① The initialization runs once at the very start. ② The condition is tested before every iteration. ③ The update runs at the end of every iteration. ④ The body is the repeated operation.

The three parts of the header run at different times (Figure 3.1.15). The order of execution is: 1 → 2 → 4 → 3 → 2 → 4 → 3 → … → 2 false → leave the loop. That is, the initialization once, then on every iteration the condition, the body, and the update. When the condition comes out false, the body does not run and the flow moves to the statement after the loop. For `for (int i = 0; i < 5; i++)` the body runs 5 times with `i` = 0, 1, 2, 3, 4. When `i` becomes 5, the condition is false. The condition is tested 6 times in all: five true, the last false.

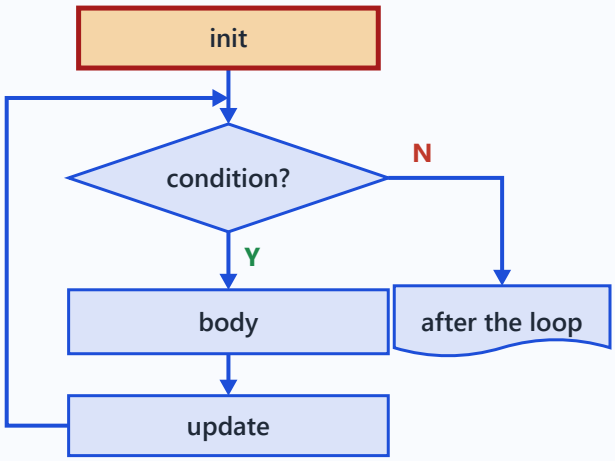


Figure 3.1.16: The flow diagram of the `for` loop. The initialization runs once; if the condition is true, the body and the update are done and the flow returns to the condition; if false, the loop is left.

Figure 3.1.16 shows the same order as a flow diagram. The operation of the loop can be summarized in six steps:

1. Initialization: the loop variable takes its initial value. This step is done once.
2. The condition is checked.
3. If the condition is true, the body runs.
4. Update: the loop variable changes.
5. The flow returns to the condition; as long as the condition is true, 3–4–5 repeat.
6. When the condition is false, the loop is left and execution continues with the next statement.

If the condition is false on the first check, the body never runs. The header `for (int i = 10; i <= 1; i++)` is valid, but its body runs zero times.

The `int i` declared in the header lives only inside the loop. Trying to read `i` after the loop gives a `cannot find symbol` error. If the last value of the loop variable is needed after the loop, the variable is declared before the loop and only its initial value is given in the header: `int i; for (i = 1; i < 6; i++) { … }`. In this case `i` is 6 when the loop ends, because the loop ends when `i < 6` becomes false.

The update does not have to be just `i++`. You can go backward with `i--`, by twos with `i += 2`, or doubling with `i *= 2`. Table 3.1.11 gives the behavior of a few headers.

Header	Values of `i`	Iterations
`for (int i = 1; i <= 5; i++)`	1, 2, 3, 4, 5	5
`for (int i = 10; i >= 1; i -= 2)`	10, 8, 6, 4, 2	5
`for (int i = 1; i <= 100; i *= 2)`	1, 2, 4, 8, 16, 32, 64	7
`for (int i = 10; i <= 1; i++)`	—	0

Header	Values of `i`	Iterations
`for (int i = 1; i != 9; i += 2)`	1, 3, 5, 7	4
`for (int i = 1; i != 10; i += 2)`	1, 3, 5, …, 9, 11, …	infinite
`for (int i = 0; i <= 10; i--)`	0, −1, −2, …	infinite*

Table 3.1.11: The values of `i` seen by the body for different `for` headers.

The last two rows need attention. The condition `i != 10` never becomes false with odd numbers: `i` jumps from 9 to 11 and never sees 10. That is why `<` or `<=` is preferred to `!=` in conditions; had `i <= 10` been written, the loop would have ended after five iterations with 1, 3, 5, 7, 9. In the last row the update goes the wrong way: as `i` decreases, `i <= 10` stays true. The starred “infinite” is meant in a practical sense. After −2,147,483,648, the smallest value of `int`, one more decrement wraps around to the largest value, 2,147,483,647, and the condition becomes false. The loop therefore ends after 2,147,483,649 iterations; with a body that prints to the screen that takes hours.

3.1.6.3 The sum from 1 to 10

The most basic use of loops is to accumulate a value step by step. The code below finds the sum of the numbers from 1 to 10:

```
int sum = 0;
for (int count = 1; count <= 10; count++) {
    sum += count;
}
System.out.println("Sum: " + sum);
```

```
Sum: 55
```

Definition 3.1.12

An **accumulator** is a variable that collects a result by having a value added to it in each iteration of a loop. It is initialized to 0 for a sum and to 1 for a product; these numbers are the identity elements of the operations.

`sum` starts at 0. The initialization step makes `count` 1. The condition `1 <= 10` is true and the first iteration runs: `sum += count`, that is, `sum = sum + count`, gives 0 + 1 = 1. The update makes `count` 2, the condition is true again, and in the second iteration `sum` becomes 1 + 2 = 3. This pattern repeats in the same way on every iteration (Table 3.1.12).

Iteration	`count`	Condition `count <= 10`	`sum` (after the body)
1	1	true	0 + 1 = 1

Iteration	`count`	Condition `count <= 10`	`sum` (after the body)
2	2	true	1 + 2 = 3
3	3	true	3 + 3 = 6
4	4	true	6 + 4 = 10
5	5	true	10 + 5 = 15
6	6	true	15 + 6 = 21
7	7	true	21 + 7 = 28
8	8	true	28 + 8 = 36
9	9	true	36 + 9 = 45
10	10	true	45 + 10 = 55
—	11	false	55 (the loop ends)

Table 3.1.12: Trace of the loop that sums 1 to 10.

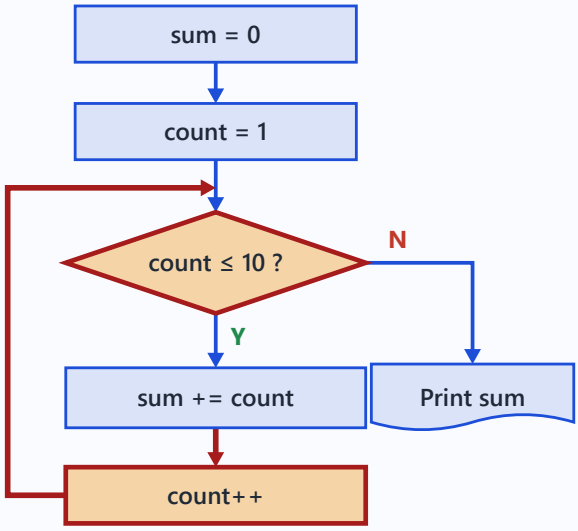


Figure 3.1.17: The flow diagram of the sum loop, at the condition test of the second iteration (`count = 2`). The bold return arrow is the path from the update to the condition.

The body runs exactly 10 times and the condition is tested 11 times. After the loop, `sum` is printed. `count`, however, is no longer defined: it lives only inside the loop. The result can also be checked with Gauss's formula: the sum of the numbers from 1 to n is $n(n + 1)/2$, and for $n = 10$ it gives 55. Comparing with a known formula is an easy way to verify the bounds of a loop.

Common mistake

Initializing the accumulator inside the loop. If the line `int sum = 0;` is moved into the body, `sum` becomes 0 again on every iteration. More importantly, the variable now lives in the body's block, and the `println` line after the loop gives a `cannot find symbol` error. The

accumulator is always declared before the loop.

3.1.6.4 Summing a range

The bounds of a loop do not have to be fixed. The program below reads two numbers from the user and adds up the integers between them:

```
import java.util.Scanner;

public class RangeSum {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Start value (a): ");
        int a = input.nextInt();
        System.out.print("End value (b): ");
        int b = input.nextInt();
        int sum = 0;
        for (int i = a; i <= b; i++) {
            sum += i;
        }
        System.out.println("Sum of the numbers: " + sum);
    }
}
```

When 3 and 7 are entered, the loop variable `i` starts from the `a` the user entered and 3 + 4 + 5 + 6 + 7 = 25 is computed:

```
Start value (a): 3
End value (b): 7
Sum of the numbers: 25
```

If the inputs are given in reverse order, the result is surprising:

```
Start value (a): 7
End value (b): 3
Sum of the numbers: 0
```

On the first check `7 <= 3` is false. The body never runs and `sum` stays 0. The program does not crash and gives no error message; it just gives a wrong answer. The fix is to put the two values in order before the loop. If `a > b`, the values of the two variables are swapped with the help of a third variable:

```
if (a > b) {
    int temp = a;
    a = b;
    b = temp;
}
```

When this block is added before the line `int sum = 0;`, the input 7 and 3 gives 25 as well. To swap the variables, writing `a = b; b = a;` is not enough: after the first assignment the old value of `a` is lost and both variables are 3. The variable `temp` is there to save this value.

3.1.7 while and do-while

3.1.7.1 The while loop

> **Definition 3.1.13**
>
> A **while loop** is a loop that tests the condition before every iteration and repeats the body as long as the condition is true. Its header contains only the condition.

```
int i = 0;
while (i < 5) {
    // statements in the loop
    i++;
}
```

`while` tests only the condition. The initialization is written before the loop and the update separately inside the body. Because the condition is tested before the body, the body may never run. When the number of iterations is not known in advance, `while` is the natural choice: “read until the user enters 0,” “drop the last digit until the number is 0,” “retry until no error is left.”

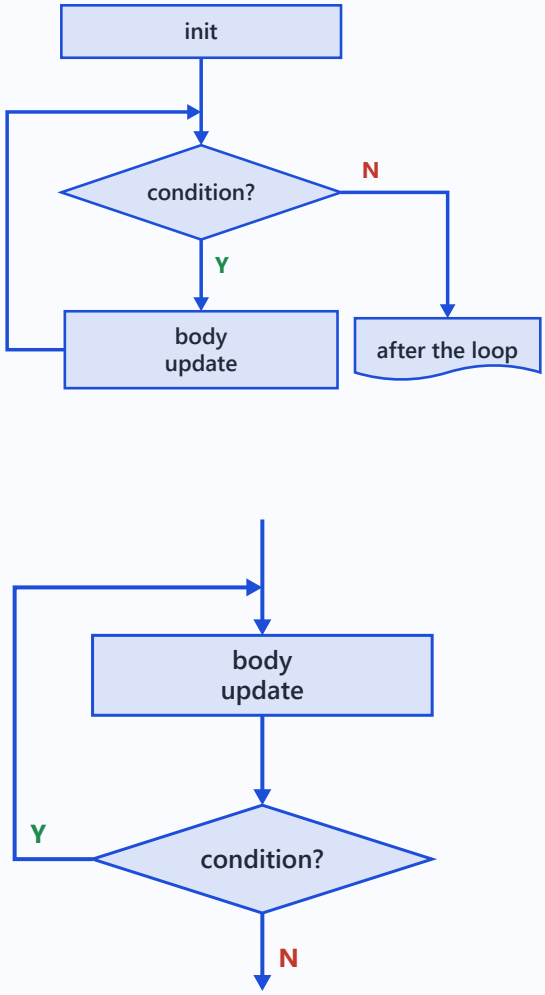


Figure 3.1.18: Left, the `while` flow diagram; right, `do-while`. In `while` the condition is in front of the body, in `do-while` behind it; in both loops the body does the update itself.

Because the update is left to the programmer, the biggest danger of `while` is the infinite loop. If `i++` is forgotten in the code above, `i` stays 0, `0 < 5` is always true, and the program never ends. The compiler does not catch this, because some programs are deliberately written with an infinite loop: the main loop of a server or an embedded system must never end.

`for` and `while` can be converted into each other. The sum from 1 to 10 can be written in either form:

```
int sum = 0;
int count = 1;
while (count <= 10) {
    sum += count;
    count++;
}
System.out.println("Sum: " + sum);
```

```
Sum: 55
```

Both loops have the same three parts: initialization `count = 1`, condition `count <= 10`, update `count++`. `for` gathers them on one line; in `while` the parts are scattered and the update can be forgotten. There is one more small difference between the two versions: in the `while` version, `count` is declared before the loop, so it can also be read after the loop, and its value is 11.

3.1.7.2 Factorial

The factorial of a positive integer is the product of all the integers from 1 to that number: 5! = 1 × 2 × 3 × 4 × 5 = 120. By definition 0! = 1. Written with `while`, it looks like this:

```
int n = 5;
int factorial = 1;
int count = 1;
while (count <= n) {
    factorial *= count;
    count++;
}
System.out.println(n + " factorial: " + factorial);
```

```
5 factorial: 120
```

The starting value of the product is 1. If it started at 0, every multiplication would give 0 and the result would always be 0. Table 3.1.13 shows each iteration of the loop.

Iteration	Condition	`factorial *= count`	`count` (after the iteration)
1	`1 <= 5` true	1 × 1 = 1	2
2	`2 <= 5` true	1 × 2 = 2	3
3	`3 <= 5` true	2 × 3 = 6	4
4	`4 <= 5` true	6 × 4 = 24	5
5	`5 <= 5` true	24 × 5 = 120	6
—	`6 <= 5` false	120 (the loop ends)	6

Table 3.1.13: Trace of the factorial loop for `n = 5`.

The factorial grows very fast and soon exceeds the limit of `int`. An `int` holds at most 2,147,483,647; 12! = 479,001,600 fits, and 13! = 6,227,020,800 does not. With `n = 13` the same code prints 1932053504. Java raises no error on integer overflow; the result is silently wrong. With `long` the results are correct up to 20!; 21! overflows `long` too.

3.1.7.3 Exponentiation

An integer power of a base is also computed by accumulating a product: $3^4 = 3 \times 3 \times 3 \times 3 = 81$. Because the number of iterations (the exponent) is known, `for` is appropriate:

```
import java.util.Scanner;

public class Power {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter base and exponent: ");
        int base = sc.nextInt();
        int exp = sc.nextInt();
        long result = 1;
        for (int i = 1; i <= exp; i++) {
            result = result * base;
        }
        System.out.println("Result: " + result);
        System.out.println("Math.pow: " + Math.pow(base, exp));
        sc.close();
    }
}
```

```
Enter base and exponent: 3 4
Result: 81
Math.pow: 81.0
```

`result` starts at 1 and the loop multiplies by `base` `exp` times. If `exp = 0`, the loop never runs and the result is correctly 1. `result` is chosen as `long` to guard against overflow: 2^{31} does not fit in an `int` but does fit in a `long`. `long` has a limit too; with the input 2 63 the program prints `Result: -9223372036854775808`.

Java's ready-made `Math.pow(base, exp)` method does the same job but returns the result as a `double`. That is why `81.0` is printed, not 81. Our own loop shows how the algorithm works. In real programs ready-made methods are preferred: they are well tested and also cover cases a loop cannot, such as fractional exponents.

3.1.7.4 The do-while loop

Definition 3.1.14

A **do-while loop** is a loop that runs the body first and then tests the condition. The body runs at least once, independent of the condition.

```
int i = 0;
do {
    // statements in the loop
    i++;
} while (i < 5);
```

`do-while` repeats as long as the condition holds; its only difference is that it tests the condition at the end of the iteration. There is one detail in the writing: the semicolon after `while (condition)` is required. If it is forgotten, the compiler gives a `';' expected` error. In a `while` loop, by contrast, that semicolon is a mistake and can produce an insidious infinite loop.

3.1.7.5 Input validation

The typical use of `do-while` is to repeat a question until a valid value is obtained from the user. The question must be asked at least once; that is exactly what `do-while` does.

```
import java.util.Scanner;

public class DoWhileInput {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int num;
        do {
            System.out.print("Enter a number (1-10): ");
            num = input.nextInt();
        } while (num < 1 || num > 10);
        System.out.println("Accepted number: " + num);
    }
}
```

Suppose the user types 15, 0, and 7 in turn:

```
Enter a number (1-10): 15
Enter a number (1-10): 0
Enter a number (1-10): 7
Accepted number: 7
```

15 and 0 are outside the range: the condition `num < 1 || num > 10` is true and the question is asked again. 7 is valid: the condition is false and the loop is left. `num` is declared before the loop because it is used both in the condition and after the loop. Had it been declared inside the body, the line `while (num < 1 …)` could not see it and would give a `cannot find symbol` error: the condition is outside the body's block.

Table 3.1.14 traces the same code with the inputs 12, −3, and 4. The short-circuiting of `||` in the condition can be seen here as well.

Iteration	`num` read	`num < 1`	`num > 10`	Condition	Result
1	12	false	true	true	the question is repeated
2	−3	true	not tested	true	the question is repeated
3	4	false	false	false	the loop is left

Table 3.1.14: Trace of the input-validation loop with the inputs 12, −3, 4.

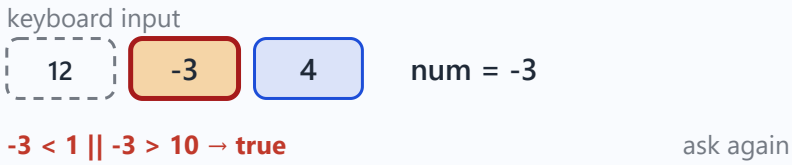


Figure 3.1.19: The input queue, second iteration. 12 has been read and rejected; now −3 has been read. Because `-3 < 1` is true, `||` does not look at the right side and the question is asked again.

The question is asked three times in all and “Accepted number: 4” is printed. The loop goes on until a valid number arrives. If the user types text such as “ten” instead of a number, `nextInt()` throws `InputMismatchException`. To handle that case too, `hasNextInt()` can be used to check first whether the input is a number.

The difference between `while` and `do-while` appears when the condition is false on the first attempt:

```
int i = 10;
while (i < 5) {
    System.out.println("while: " + i);
    i++;
}
int j = 10;
do {
    System.out.println("do-while: " + j);
    j++;
} while (j < 5);
System.out.println("i = " + i + ", j = " + j);
```

```
do-while: 10
i = 10, j = 11
```

With `i = 10`, `while` tests the condition first: `10 < 5` is false and the body never runs; `i` stays 10. With `j = 10`, the `do` block is entered without looking at the condition. "do-while: 10" is printed and `j` becomes 11. Only now is the condition tested: `11 < 5` is false and the loop is left. The same condition, a different result: `while` ran 0 iterations and `do-while` ran 1.

A real consequence of this difference appears in counting digits. The loop `while (num > 0) { digits++; num /= 10; }`, which counts the digits of a number, gives 4 for 2024 (2024 → 202 → 20 → 2 → 0), but 0 for 0: the condition is false on the first attempt. Yet 0 is a one-digit number. When the loop is converted to `do-while`, the body runs at least once and 1 is found for 0:

```
int num = 0, digits = 0;
do {
    digits++;
    num /= 10;
} while (num != 0);
System.out.println(digits);
```

The condition is written here as `num != 0`. That way negative numbers are counted correctly too: 2 is found for −45. With `while (num > 0)` the loop never runs for negative numbers.

3.1.7.6 Comparing the three loops

	`for`	`while`	`do-while`
When is the condition tested?	Before every iteration	Before every iteration	After every iteration
Minimum iterations?	0	0	1
Initialization / update	In the header, on one line	Outside the loop / in the body	Outside the loop / in the body
When to use?	The number of iterations is known: "1 to *n*," "every element of the array"	Until a condition changes: "until the number is 0"	The task must be done at least once: input validation, a menu
Syntax	`for (init; condition; update) { }`	`while (condition) { }`	`do { } while (condition);`

Table 3.1.15: Comparison of the `for`, `while`, and `do-while` loops.

The three loops can be converted into one another; the choice is made by readability. If a loop advances with a counter, `for` is written: someone reading the header can see how many times the loop will run. If the end of the loop depends on an event, `while` is clearer. If the body must run at least once, `do-while` is chosen.

3.1.7.7 Infinite loops and boundary errors

Most loop errors fall into two groups: loops that never end, and loops that run one iteration too many or too few. The compiler catches neither group.

An **infinite loop** is a loop whose condition never becomes false. The most common causes are: forgetting the update in `while`, making the update in the wrong direction (`i--` with `i <= 10`), and a `!=` condition that skips over the target value. A more insidious cause is a semicolon placed after the loop header:

```
int i = 0;
while (i < 5); {
    i++;
}
```

The semicolon sets up a loop with an empty body. `i` never increases, `0 < 5` is always true, and the program silently freezes. The block below does not belong to the loop; the flow never reaches it. If the same mistake is made in a `for` header, no infinite loop results but the logic is still broken: `for (int i = 0; i < 3; i++); { System.out.println("hello"); }` runs the empty body three times and prints “hello” only once.

> **Definition 3.1.15**
>
> An **off-by-one error** is a mistake, usually in the starting value or in the comparison operator of the condition, that makes a loop run one iteration too many or one too few.

For “1 to 10,” writing `i < 10` skips 10 and runs 9 iterations. With `i = 0` and `i <= 10`, it runs 11 iterations. These mistakes are easy to find by tracing by hand: the value of the loop variable on the first iteration and its value on the last are checked separately. How many iterations a loop runs can also be computed with the formula *end − start + 1* (with `<=`, in steps of one). A typical example is a pattern in which every line is “+-” repeated 10 times. The line is 20 characters; if the inner loop starts at `j = 0` and runs with `j < 21`, 21 characters are printed and the line ends with an extra “+.”

Keeping a counter with floating-point numbers is also a source of boundary errors. The loop `for (double x = 0; x != 1.0; x += 0.1)` never ends: adding 0.1 ten times gives not exactly 1.0 but 0.9999999999999999, and `x` skips over 1.0. Loop counters should always be integers; if a decimal value is needed, it is computed from the counter in the body: `double x = i * 0.1;`.

3.1.8 Nested loops

Definition 3.1.16

A **nested loop** is a loop inside the body of another loop. The outer loop is called the **outer loop** and the one in the body the **inner loop**.

Nested loops are used when working with two-dimensional structures: tables, matrices, patterns drawn on the screen, the pixels of an image. The analogy is a clock. Each time the hour hand (the outer loop) moves one step, the minute hand (the inner loop) makes a full turn. The hour hand does not move until the minute hand has finished its turn.

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 2; j++) {
        // nested loop body
    }
}
```

On every iteration of the outer loop the inner loop starts over and runs to its end. The total number of times the body runs is therefore the product of the iteration counts of the two loops: outer 3 iterations × inner 2 iterations = 6 times (Figure 3.1.20).

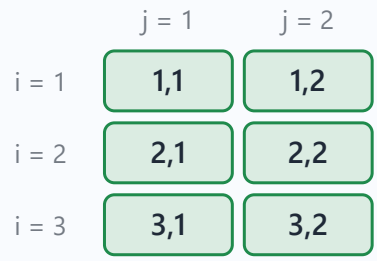


Figure 3.1.20: A grid of 3 rows and 2 columns. Each cell is one run of the body: 6 in all.

The product rule also shows the cost of nested loops. Two loops of 1,000 iterations each run the body 1,000,000 times, and three loops 1,000,000,000 times. A billion operations can take a few seconds on today's computer. As the number of nested loops grows, the code becomes both more complicated and slower. How many nested loops a problem needs is the main factor that determines the speed of the algorithm.

3.1.8.1 How a nested loop works

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 2; j++) {
        System.out.println("i = " + i + ", j = " + j);
    }
}
```

The flow goes step by step as follows. The outer loop starts: `i = 1`, `1 <= 3` is true. The inner loop starts: `j = 1` and the body prints "i = 1, j = 1." With `j++`, `j` becomes 2 and "i = 1, j = 2" is printed. When `j` becomes 3, `3 <= 2` is false and the inner loop is complete. The flow returns to the outer loop: with `i++`, `i` becomes 2, the condition is true, and the inner loop starts **again** from `j = 1`. This continues until the `i = 3` iteration is finished too. Finally `i` becomes 4, `4 <= 3` is false, and the outer loop ends as well.

Step	`i`	`j`	Event	Screen
1	1	1	the inner loop starts	i = 1, j = 1
2	1	2		i = 1, j = 2
3	1 → 2	3	`3 <= 2` false: the inner loop ends, `i++`	
4	2	1	the inner loop starts again	i = 2, j = 1
5	2	2		i = 2, j = 2
6	2 → 3	3	the inner loop ends, `i++`	
7	3	1	the inner loop starts again	i = 3, j = 1
8	3	2		i = 3, j = 2
9	3 → 4	3	the inner loop ends; `4 <= 3` false: the outer loop ends	

Table 3.1.16: Trace of a nested loop.

The body runs 3 × 2 = 6 times. The outer condition is tested 4 times and the inner condition 3 × 3 = 9 times: each time the inner loop starts it makes two true tests and one false test. In the output you can see that `i` changes slowly and `j` quickly: while `i` is 1, `j` goes over the values 1 and 2.

3.1.8.2 A star pattern

The bound of the inner loop does not always have to be fixed. If the bound is tied to the outer loop's variable, a different number of iterations is made on each row:

```
for (int i = 1; i <= 4; i++) {
    for (int j = 1; j <= i; j++) {
        System.out.print("* ");
    }
    System.out.println();
}
```

```
*
* *
* * *
* * * *
```

The condition of the inner loop is `j <= i`. In the first row `i = 1`, so the inner loop runs once; in the second row twice, in the third three times, and in the fourth four times. `print("* ")` does not move to a new line; the stars line up side by side. When the inner loop ends, `println()` moves to the next line. This line is outside the inner loop and inside the outer loop: it runs once at the end of each row.

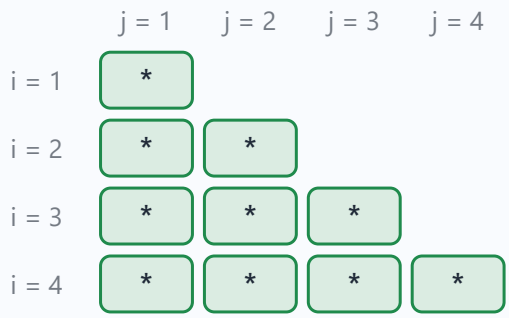


Figure 3.1.21: The finished star pattern. In row `i`, the cells `j` = 1 … `i` are filled.

The total number of stars is 1 + 2 + 3 + 4 = 10. In general, a triangle of n rows prints $n(n + 1)/2$ stars. This number is the typical cost of loops whose inner bound depends on the outer one: about half of the n^2 of a rectangular grid. If the inner loop started at `j = i` and ran to 4, the iteration counts would be 4, 3, 2, 1 and the total would again be 10; this time the pattern would be an inverted triangle.

Common mistake

Putting `println()` at the wrong level. If it is written inside the inner loop, a new line follows every star and a single column of ten lines results. If it is written outside the outer loop, all the stars line up on a single line. Which line belongs to which loop is determined only by braces; keeping the indentation tidy makes this relationship visible.

3.1.8.3 Place value

```
for (int tens = 0; tens <= 4; tens++) {
    for (int ones = 1; ones <= 9; ones++) {
        int num = tens * 10 + ones;
        System.out.println("Tens digit: " + tens
                + ", ones digit: " + ones + " --> " + num);
    }
}
```

This code builds two-digit numbers from a tens digit and a ones digit. To find how many lines it prints, the iteration counts of the two loops are multiplied. The outer loop runs 5 iterations for `tens` = 0 … 4 and the inner loop 9 iterations for `ones` = 1 … 9: 5 × 9 = 45 lines. The first and last lines are:

```
Tens digit: 0, ones digit: 1 --> 1
Tens digit: 0, ones digit: 2 --> 2
Tens digit: 0, ones digit: 3 --> 3
…
Tens digit: 4, ones digit: 8 --> 48
Tens digit: 4, ones digit: 9 --> 49
```

`num = tens * 10 + ones` takes values from 1 to 49. Because `ones` does not start at 0, 10, 20, 30, and 40 are never printed; 0 is not printed either. There are 49 numbers from 1 to 49, and since 4 of them are skipped, 45 lines remain. The two computations confirm each other.

The print line of the code is split over two lines. In Java a long string can be broken into pieces with `+` and continued on the next line. Text inside quotation marks, however, cannot be cut at the end of a line. Writing `"Tens digit:` and ending the line gives an `unclosed string literal` error.

3.1.8.4 The multiplication table

```
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 10; j++) {
        int product = i * j;
        System.out.print(i + "x" + j + "=" + product + "\t");
    }
    System.out.println(); // new line
}
```

The outer loop goes over the rows (`i`) and the inner loop over the columns (`j`): 10 × 10 = 100 products. `print` does not move to a new line; the `"\t"` added after each product is a tab character, which aligns the columns to the next tab stop. When the row is finished, `println()` moves to the next line.

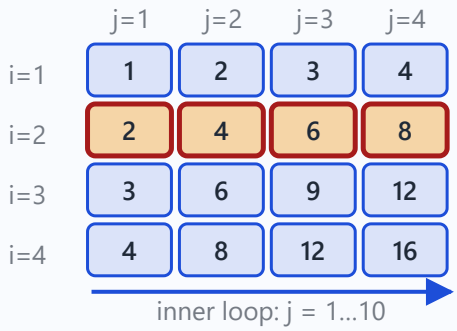


Figure 3.1.22: The 4 × 4 corner of the multiplication table. The highlighted row is the `i = 2` iteration: the inner loop goes over the columns with `j` = 1, 2, 3, 4 ….

```
1x1=1   1x2=2   1x3=3   1x4=4   1x5=5   1x6=6   …
2x1=2   2x2=4   2x3=6   2x4=8   2x5=10   2x6=12   …
3x1=3   3x2=6   3x3=9   3x4=12   3x5=15   3x6=18   …
…
10x1=10   10x2=20   10x3=30   10x4=40   10x5=50   …
```

Tab stops are usually 8 characters apart. A cell of 9 characters such as “10x10=100” overshoots a stop and can shift the rest of the row. For a tidier table, fixed-width fields are used with `printf`: `System.out.printf("%4d", i * j);` writes each number right-aligned in a field of 4 characters. `%-4d` left-aligns, and `%04d` fills missing digits with 0.

3.1.9 break and continue

Loops normally end when the condition becomes false. Sometimes a decision has to be made in the middle of a loop: if the thing being sought has been found, the remaining iterations are pointless; if an item is useless, that iteration should be skipped. Java offers two statements for these two jobs.

Definition 3.1.17

`break` ends the nearest enclosing loop (or `switch`) at once; the flow moves to the statement after the loop. `continue` skips the rest of the current iteration and moves to the next one; in `for`, the update runs first and then the condition.

3.1.9.1 break in loops

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        break;
    }
    System.out.println("Loop step: " + i);
}
System.out.println("Left the loop.");
```

```
Loop step: 1
Loop step: 2
Loop step: 3
Loop step: 4
Left the loop.
```

For `i` = 1, 2, 3, and 4, `i == 5` is false; `break` is skipped and the step is printed. At `i = 5` the condition is true and `break` runs. The loop ends at that moment: 5 is not printed, and 6–10 are never tried. The flow moves to the line after the loop and "Left the loop." is printed.

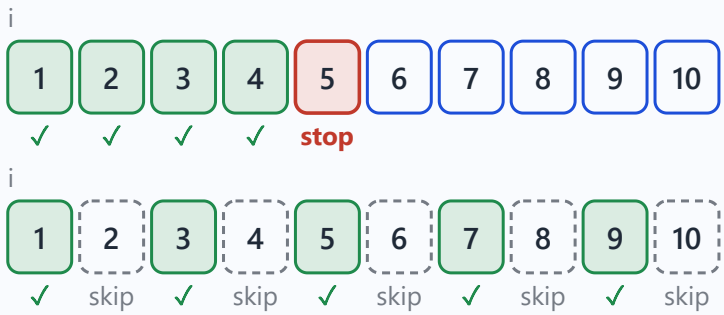


Figure 3.1.23: Top, `break`: 1–4 are printed, the loop stops at 5, and 6–10 are never processed. Bottom, `continue`: the odd numbers are printed and the even numbers skipped, but the loop goes on to 10.

A statement written right after `break` in the same block can never be reached. Java treats this as a compile-time error: the code `for (…) { break; System.out.println(i); }` gives `error: unreachable statement`. That is why `break` is almost always used inside an `if`.

3.1.9.2 Prime numbers: a flag and break

Finding out whether a number is prime is the best-known use of `break`. A prime number is an integer greater than 1 that is divisible only by 1 and itself. The program looks for a divisor starting from 2; the moment it finds the first divisor, it stops the search.

```java
import java.util.Scanner;

public class PrimeCheck {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int num = sc.nextInt();
        boolean isPrime = true;            // flag
        if (num < 2) isPrime = false;     // 0 and 1 are not prime
        for (int divisor = 2; divisor <= num / 2; divisor++) {
            if (num % divisor == 0) {      // divides evenly
                isPrime = false;
                break;                    // break out of the loop
            }
        }
        if (isPrime) {
            System.out.println(num + " is prime.");
        } else {
            System.out.println(num + " is not prime.");
        }
        sc.close();
    }
}
```

```
Enter a number: 21
21 is not prime.

Enter a number: 97
97 is prime.
```

`isPrime` is a **flag**: a `boolean` variable that carries whether a condition has occurred during the loop. At first the number is assumed to be prime. Since 0 and 1 are not prime, the flag is lowered at once when `num < 2`. The loop tries the divisors from 2 to `num / 2`. If an exact divisor is found, the flag is lowered and the loop is left with `break`. After the loop the flag is examined and the result is printed.

`num`	**Divisors tried**	**Attempts**	`isPrime`
21	2 (remainder 1), 3 (remainder 0) → `break`	2	`false`
97	2, 3, ..., 48; none divides	47	`true`
91	2, 3, 4, 5, 6, 7 (remainder 0) → `break`	6	`false`
2	none (`2 <= 1` false)	0	`true`
1	none; the flag is lowered by `num < 2`	0	`false`

Table 3.1.17: Trace of the prime-number program with a few inputs.

97 is prime, and the program finds this out only after 47 attempts. There is a faster way. If a number has a divisor other than 1 and itself, one of those divisors is less than or equal to the square root of the number. This is because if $num = a \times b$, a and b cannot both be greater than $\sqrt{num}$; their product would exceed num. The loop condition is changed accordingly:

```
for (int divisor = 2; divisor * divisor <= num; divisor++) {
    if (num % divisor == 0) {
        isPrime = false;
        break;
    }
}
```

The condition `divisor * divisor <= num` applies the square-root bound without calling `Math.sqrt` and without carrying floating-point error. For 97, $\sqrt{97} \approx 9.85$; the divisors 2 … 9 are tried, and only 8 attempts are made instead of 47. For a seven-digit prime such as 1,000,003 the difference is much greater: 500,000 attempts with the `num / 2` bound and 999 with the square-root bound.

3.1.9.3 continue in loops

```
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0) {
        continue;
    }
    System.out.println("Odd number: " + i);
}
```

```
Odd number: 1
Odd number: 3
Odd number: 5
Odd number: 7
Odd number: 9
```

`i = 1` is odd, the condition is false, and the number is printed. `i = 2` is even: `continue` runs, the rest of the body, that is, the print line, is skipped, and the flow goes directly to `i++`. This continues up to 10. When `i = 11`, `11 <= 10` is false and the loop ends normally. `continue` does not end the loop; it only cuts that iteration short. The same job can also be written without `continue` by reversing the condition: `if (i % 2 != 0) System.out.println(…);`. `continue` flattens the code when the cases to be skipped are to be filtered out at the start of the body, that is, when the body is long.

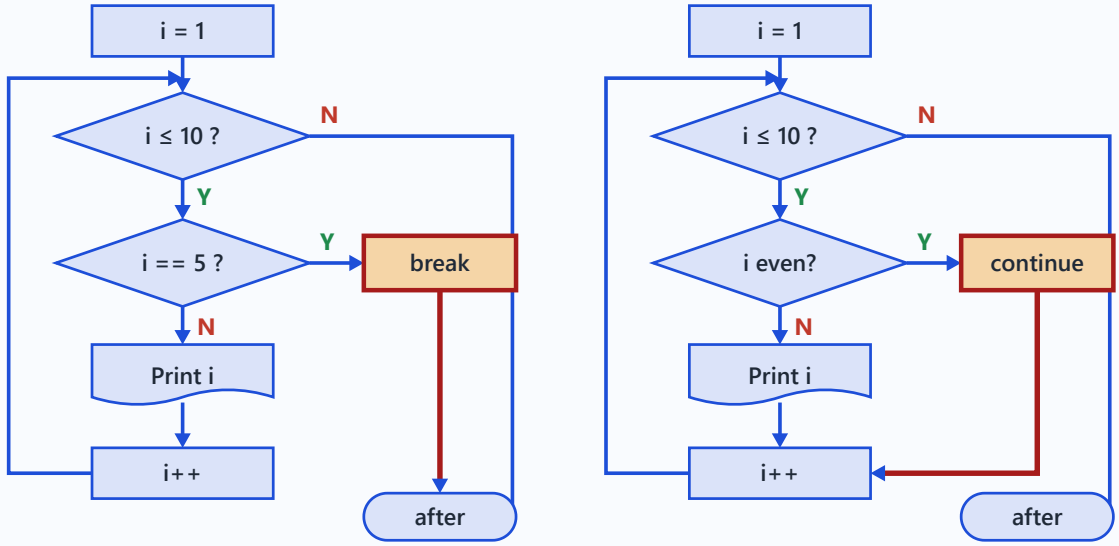


Figure 3.1.24: Left, `break` : when `i == 5` the flow leaves the loop completely. Right, `continue` : when `i` is even the print is skipped and the flow goes to the update (`i++`); the loop goes on.

`continue` has a hidden danger in a `while` loop. In `for` , the update is in the header and runs after `continue` as well. In `while` , the update is inside the body, and `continue` can skip it too:

```
int i = 0;
while (i < 5) {
    if (i == 2) {
        continue;
    }
    System.out.print(i + " ");
    i++;
}
System.out.println("done");
```

This code prints “0 1 ” and then loops forever. When `i` becomes 2, `continue` skips the rest of the body, including `i++` . `i` stays 2, the condition is always true, and “done” is never printed. If the loop is converted to the form `for (int i = 0; i < 5; i++)` , the update moves to the header and the output is “0 1 3 4 done.”

3.1.9.4 break and labels in nested loops

`break` ends only the **nearest** loop it is in. A `break` in the inner loop does not touch the outer loop:

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (j == 2) {
            break;
        }
        System.out.println(i + " " + j);
    }
}
```

```
1 1
2 1
3 1
```

For each `i` the inner loop prints `j = 1` and ends with `break` at `j = 2`. The outer loop goes on and starts the inner loop again with the next `i`. In the end only `j = 1` is printed for each `i`.

If both loops have to be left at once, Java offers the **labeled** `break`. A name and a colon are written in front of the outer loop; `break` is used with that name:

```
outer:
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (i * j == 4) {
            break outer;
        }
        System.out.println(i + " x " + j);
    }
}
System.out.println("done");
```

```
1 x 1
1 x 2
1 x 3
2 x 1
done
```

The label `outer:` names the outer loop. In the `i = 1` iteration the three products (1, 2, 3) are not equal to 4 and all are printed. The inner loop ends and the outer loop moves to `i = 2`. `2 × 1 = 2` is printed. At `i = 2`, `j = 2`, `2 × 2 = 4` and the condition is true. `break outer;` leaves not only the inner loop but the labeled outer loop as well. The remaining cells (`2 × 3` and the whole `i = 3` iteration) are never visited. The flow jumps past both loops and “done” is printed.

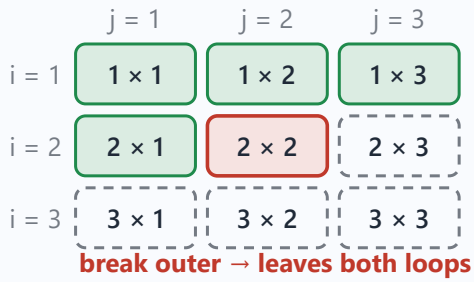


Figure 3.1.25: The labeled `break`. The green cells have been printed; at the `2 × 2` cell the condition came out true and `break outer` left both loops at once. The dashed cells were never visited.

With an unlabeled `break`, only the inner loop would be left and the `i = 3` iteration would run too. The products in that iteration are 3, 6, and 9; none equals 4, so all three are printed. The output would be seven lines: 1 x 1, 1 x 2, 1 x 3, 2 x 1, 3 x 1, 3 x 2, 3 x 3, followed by “done.” `continue` can also be used with a label. `continue outer;` abandons the rest of the inner loop and moves to the next iteration of the outer loop:

```
outer:
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (j == 2) {
            continue outer;
        }
        System.out.println(i + " " + j);
    }
}
```

```
1 1
2 1
3 1
```

Labels are rarely used. Because they carry the flow several levels away, they make the code harder to follow. A more common way of leaving two loops is to move the nested loop into a separate method and return from the method with `return` when the sought value is found. In Java `goto` is a reserved word but cannot be used; the labeled `break` and `continue` are the only forms of “jumping” the language allows.

3.1.10 Common mistakes

Some of the mistakes in control structures are caught by the compiler, and its message points to the place. Others compile, run, and merely give a wrong result. The second group is more dangerous, because nothing warns the programmer. Table 3.1.18 collects the typical mistakes in decision

structures and Table 3.1.19 those in loops. The compiler messages were obtained with JDK 21.

Mistake	Example	Result	Correct form
Comparing with `=`	`if (x = 5)`	`incompatible types: int cannot be converted to boolean`	`if (x == 5)`
Semicolon after `if`	`if (x > 5); { … }`	Compiles; the block always runs	Delete the semicolon
Block without braces	`if (k) a(); b(); else c();`	`'else' without 'if'`	`{ }` for several statements
Block without braces, no `else`	`if (k) a(); b();`	Compiles; `b()` always runs	`if (k) { a(); b(); }`
Writing a range as in mathematics	`60 <= grade < 90`	`bad operand types for binary operator '<'`	`grade >= 60 && grade < 90`
Wrong order in a chain	`if (grade >= 60) … else if (grade >= 90) …`	Compiles; the 90 branch is never reached	Arrange thresholds from largest to smallest
Forgetting `break`	`case 2: … case 3: …`	Compiles; the `case` labels below run too	End every `case` with `break`
Assignment without `default`	The variable is assigned only in the `case` labels	`variable season might not have been initialized`	Add `default`
A variable as a `case` label	`int b = 2; … case b:`	`constant expression required`	A constant or a `final` variable
A period in decimal input	`-7.5` for `nextDouble()` in the Turkish locale	`InputMismatchException`	Type `-7,5` or use `useLocale`

Table 3.1.18: Common mistakes in decision structures.

Mistake	Example	Result	Correct form
Forgetting the update	`while (i < 5) { println(i); }`	Infinite loop; the compiler does not catch it	`i++` in the body
Updating in the wrong direction	`for (int i = 0; i <= 10; i--)`	Practically an infinite loop	`i++`
A semicolon after the loop header	`while (i < 5); { i++; }`	An infinite loop with an empty body	Delete the semicolon
Off-by-one error	`i < 10` for "1 to 10"	Compiles; runs 9 iterations	`i <= 10`; check the first and last iteration
Using the loop variable outside	`for (int i …) { }` then `println(i)`	`cannot find symbol`	Declare `i` before the loop
Code after `break`	`break; System.out.println(i);`	`unreachable statement`	Move the line before `break`
Wrong starting value	`int factorial = 0;`	Compiles; the result is always 0	A sum starts at 0, a product at 1
Resetting the accumulator in the loop	`int sum = 0;` in the body	Reset on every iteration; not accessible outside the loop	Declare it before the loop
`continue` in `while`	`if (i == 2) continue;` followed by `i++`	Infinite loop	Use `for`, or do `i++` first

Mistake	Example	Result	Correct form
No semicolon at the end of `do-while`	`} while (i < 3)`	`';' expected`	`} while (i < 3);`
`int` overflow	13! with `int`	Compiles; prints 1932053504	`long`

Table 3.1.19: Common mistakes in loops.

The most reliable way to find mistakes that compile is to trace by hand. The code is run on paper with a few small inputs, writing the variables into a table after each iteration. Three inputs in particular should be tried: the case in which the loop should not run at all (e.g. `n = 0`), the case in which it should run once, and a boundary value. A trace table showing the variables after each iteration reveals boundary errors without running the code.

3.1.11 Summary

- **Three basic structures:** sequential execution, decisions, and loops. Every algorithm can be written by combining them. In sequential code each line depends on the state left by the previous lines.
- **Condition:** a `boolean` expression whose result is `true` or `false`. `=` is assignment and `==` is comparison; Java does not compile `if (x = 7)`.
- `if` / `else`**:** if the condition is true one block runs, otherwise the other; at most one of the two blocks runs. Without braces only the first statement is attached to the `if`; indentation means nothing.
- **The** `else if` **chain:** conditions are tried from top to bottom, the block of the first true condition runs, and the chain ends. The order of the conditions takes the place of upper bounds.
- **Nested decisions:** the inner condition is tested only if the outer condition is true. An `else` attaches to the nearest `if` that has no `else` yet. Deep structures are flattened with `&&`.
- **Short-circuiting:** `&&` does not evaluate the right side if the left is `false`, and `||` does not if the left is `true`. `b != 0 && a / b > 2` is safe; operations with side effects are not written on the right. `?:` is an expression that chooses one of two values.
- `switch`**:** compares a value with constant `case` labels. A `case` is an entry point; without `break` the flow falls into the `case` labels below. The selector can be `byte`, `short`, `char`, `int`, their wrappers, `String`, or `enum`; labels are constant and unique.
- **The** `switch` **expression (Java 14+):** the form `case 3, 4, 5 -> …`; there is no fall-through, it produces a value, and it must cover all possibilities (`default` is required for `int`).
- `for`**:** initialization, condition, and update on one line; the order of execution is initialization, then on every iteration condition → body → update. A variable declared in the header lives only in the loop.

- `while` **and** `do-while`**:** `while` tests the condition first and may run 0 iterations; `do-while` tests afterwards, runs at least 1 iteration, and needs a `;` at the end. Input validation is the typical use of `do-while`.
- **Accumulator:** a sum starts at 0, a product at 1, and it is declared before the loop. `int` is enough up to 12!, and `long` up to 20!; overflow is silent.
- **Nested loops:** the inner loop starts over on every iteration of the outer loop; the body runs as many times as the product of the iteration counts. If the inner bound depends on the outer (`j <= i`), the total is *n(n + 1)/2*.
- `break` **and** `continue`**:** `break` ends the nearest loop, and `continue` skips the rest of the iteration. The labeled `break outer;` leaves the named outer loop too. A flag and `break` are used together in the prime-number search; the bound is shortened with `divisor * divisor <= num`.
- **Mistakes:** infinite loops, off-by-one errors, a wrong order of conditions, and a forgotten `break` compile but run wrongly; they are found by tracing by hand.

3.1.12 Key terms

control structure condition block if statement else-if chain nested decision
short-circuit evaluation conditional operator switch statement fall-through switch expression
loop for loop while loop do-while loop accumulator nested loop break / continue iteration
infinite loop off-by-one error flag labeled break

3.1.13 Test Yourself

1. What does the chain below print for `temp` = 18, 15, 0, and −3?

```
if (temp < 0) System.out.println("freezing");
else if (temp < 15) System.out.println("cold");
else if (temp < 25) System.out.println("mild");
else System.out.println("hot");
```

Answer

18: mild; 15: mild; 0: cold; −3: freezing. In a chain written with `<`, the thresholds go from smallest to largest. 15 passes the “cold” link because `15 < 15` is false, and falls to “mild.” 0 does not give “freezing” either, because `0 < 0` is false.

2. The line `boolean s = (a != 0) && (b / a > 1) || b > 3;` is run with `int a = 0, b = 5;`.

Does the program give a division-by-zero error? What is `s`?

Answer

It does not; `s = true`. `&&` comes before `||`; the expression reads `((a != 0) && (b / a > 1)) || (b > 3)`. Because `a != 0` is false, `b / a` is never evaluated, thanks to short-circuiting, and the parenthesized part is `false`. Then `b > 3` comes out true and the result is `true`.

3. What value does the following `switch` give `t` for `k` = 3, 1, 5, and 7?

```
int t = 0;
switch (k) {
    case 1: t += 1;
    case 3: t += 3;
    case 5: t += 5; break;
    default: t += 100;
}
```

Cevap

`k = 3`: 3 + 5 = 8. `k = 1`: 1 + 3 + 5 = 9. `k = 5`: 5. `k = 7`: 100. Because the first two `case` labels have no `break`, the flow falls through to `case 5`; the `break` in `case 5` prevents the move into `default`.

4. How many iterations does the loop `int i; for (i = 3; i <= 30; i += 4) { … }` run? Which values of `i` does the body see? What is `i` after the loop?

Answer

7 iterations: 3, 7, 11, 15, 19, 23, 27. The next value is 31 and `31 <= 30` is false; after the loop `i = 31`. The number of iterations can also be computed as ⌊(30 − 3) / 4⌋ + 1 = 6 + 1 = 7.

5. After the code `int n = 100, steps = 0; while (n > 1) { n /= 2; steps++; }`, what are `steps` and `n`?

Answer

`steps = 6`, `n = 1`. `n`: 100 → 50 → 25 → 12 → 6 → 3 → 1. Integer division discards the fraction: 25 / 2 = 12, 3 / 2 = 1. The loop finds the number of halvings needed to bring the number down to 1, that is, $\lfloor \log_2 100 \rfloor = 6$.

6. After the code `int x = 0; do { x += 3; } while (x < 0);` and `int y = 0; while (y < 0) {`

`y += 3; }`, what are `x` and `y`?

Answer

`x = 3`, `y = 0`. The condition is false on the first test in both loops. `do-while` runs the body once without looking at the condition; `while` tests the condition first and never enters the body.

7. How many times does the body run in the code below, and what is `count`?

```
int count = 0;
for (int i = 1; i <= 5; i++)
    for (int j = 1; j <= i; j++)
        if ((i + j) % 2 == 0) count++;
```

Cevap

The body runs 1 + 2 + 3 + 4 + 5 = 15 times. The pairs whose sum is even number 1, 1, 2, 2, 3 for `i` = 1 … 5 respectively: `count = 9`. For example, at `i = 5` the sums for `j` = 1, 3, 5 are 6, 8, 10.

8. What does the following code print?

```
int s = 0;
for (int i = 1; i <= 10; i++) {
    if (i % 4 == 0) continue;
    if (i > 7) break;
    s += i;
}
System.out.println(s);
```

Cevap

24. 1, 2, 3 are added (6). 4 is skipped by `continue`. 5, 6, 7 are added (24). 8, being divisible by 4, is again skipped by `continue`; it does not reach the `break` line. At 9, `9 > 7` is true and `break` ends the loop; 9 and 10 are not added.

9. Which of these compile? (a) `switch (3.0) { default: }` (b) `final int K = 2; switch (x) {`

`case K: break; }` (c) `if (x = 0) { }` (d) `while (true) { } System.out.println();`

Answer

Only (b). (a) `selector type double is not allowed`. (b) `K` is a `final` variable with a constant value and can be a label. (c) `incompatible types: int cannot be converted to boolean`. (d) Because there is no `break` inside the `while (true)` loop, it never ends, and an `unreachable statement` error is given for the next line.

3.1.14 Further reading

- Oracle, *The Java Tutorials*, "Control Flow Statements": a short, official account with examples of `if`, `switch`, loops, and `break` / `continue`; there is also a separate "Switch Expressions" page for switch expressions.
- A. B. Downey and C. Mayfield, *Think Java* (2nd ed., 2019), Chapters 5–6: conditions, logical operators, and loops are treated in plain language with many short exercises. The book can be read free online.
- J. Gosling et al., *The Java Language Specification, Java SE 21 Edition*, Chapter 14: the exact rule for every statement, fall-through, labels, and the full definition of the "unreachable statement" check are here.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals* (12th ed.), Section 3.8: explains control flow by comparison with C/C++; a short reference for the experienced reader.

3.1.15 References

1. C. Böhm, G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", *Communications of the ACM*, 9(5), 366–371, 1966.
2. P. Deitel, H. Deitel, *Java: How to Program (Early Objects)*, 11th ed., Pearson, 2017 (Chapters 4–5: Control Statements).
3. A. B. Downey, C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019 (Chapter 5: Conditionals and Logic; Chapter 6: Loops).
4. J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, G. Bierman, *The Java Language Specification, Java SE 21 Edition*, Oracle, 2023 (Chapter 14: Blocks, Statements, and Patterns).
5. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson, 2022 (3.8: Control Flow).
6. Oracle, *The Java Tutorials: Control Flow Statements*, docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html.
7. K. Sierra, B. Bates, T. Gee, *Head First Java*, 3rd ed., O'Reilly, 2022.

PART I · CHAPTER 3.2

Methods

As a program grows, it no longer fits inside a single `main` method. The same calculation is needed in several places, and bugs get lost in a long pile of code. The method is the answer to both problems: write a job once, give it a name, and call it as often as you like. In Java every function lives inside a class and is called a method. These notes show how a method is defined and called, what happens in memory during a call, and why arguments are copied. They also cover how the compiler chooses between methods with the same name, the built-in Java methods, and the scope and lifetime of variables. The last sections discuss how to design a good method. Every program in the examples was compiled and run with JDK 21; the values in the trace tables and the compiler messages come from those runs.

LEARNING OUTCOMES

Defining and calling
Write a method with an access modifier, a return type, a name and a parameter list; call it with arguments and use the value it returns.

Tracing execution
Show step by step how stack frames are pushed and popped on the call stack and how arguments are copied by value.

Overloading
Tell apart methods that share a name by their parameter lists, and predict from the argument types which version the compiler will pick.

Built-in methods
Use the methods of `System.out` and `Math` correctly, and recognize the commonly used classes and packages of the Java library.

Scope
Explain the scope and lifetime of local variables, class fields and nested blocks; spot shadowing.

Design
Design good methods using abstraction, generalization, DRY and the single-responsibility principle; split a problem into methods.

3.2.1 The function concept

So far every program has consisted of a single method: `main`. For short programs that is enough. But consider a student grade system. Grades are read, the average is computed, a letter grade is found, and the results are printed. If all of these steps are written one after another inside `main`, the result is one block of several hundred lines. If the average formula is needed in three places, it gets copied three times, and when the formula turns out to be wrong, all three copies must be fixed separately. Methods remove this mess.

3.2.1.1 What is a function?

> **Definition 3.2.1**
>
> A **function** is a named piece of code that does one job and can be used again and again. It is defined by the parameters it takes and the type of the value it returns. In Java every function is written inside a class and is called a **method**.

A function is like a recipe. “Brew tea” is a recipe: it is written once and followed as often as needed. The ingredients are the inputs of the recipe; their counterpart in programming is the **parameters**. The tea that comes out is the result of the recipe; its counterpart is the **return value**. Whoever follows the recipe does not rethink every step; they just say “brew tea” and get the result. A program does the same: it calls a method by name and uses its result.

A function is not really a new idea. `main`, which every Java program has, is a method. `System.out.println(...)`, used to print text, is a call to a method that comes with the Java library. The programmer does not know how `println` sends characters to the screen; they only know what it does. These notes show how to build the same structure with your own methods.

The benefits of using methods can be listed as follows:

- **Readability.** A well-named method call says in one line what is being done. Someone reading `computeAverage(grades)` understands what happens without looking at the details of the loop.
- **Less duplicated code.** The same job is written once and called whenever it is needed. When a bug appears, only one place has to be fixed.
- **Modularity.** The program is split into small, manageable pieces. Each piece can be written and tested separately.
- **Structure.** The links between the pieces are limited to parameters and return values. When the inside of one piece changes, the others are not affected.

In these notes the words “function” and “method” are mostly used interchangeably. The distinction is this: a function is the general programming concept, and a method is its form in Java. In languages such as C, functions can be written without belonging to any class. In Java, a method cannot be

written outside a class.

3.2.1.2 Input, process, output

Every function can be thought of in three stages. In the **input** stage the function is given data; these data are the parameters, and they determine what the function works on. In the **process** stage the statements in the body of the function work on the input. In the **output** stage a result is produced and handed back to the caller with the `return` statement. Figure 3.2.1 shows the three stages for the `add` function.

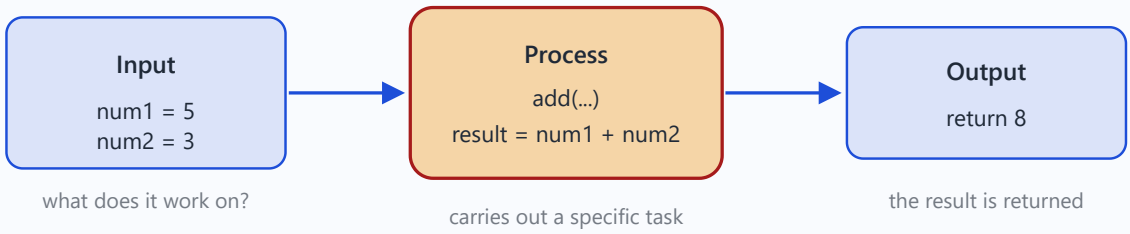


Figure 3.2.1: Input, process and output. The inputs (`num1 = 5` , `num2 = 3`) enter the `add` function. The body performs `result = num1 + num2` . `return 8` hands the result back to the caller.

This diagram is the same as the function concept in mathematics. In the function $f(x) = x^2$, x is the input, squaring is the process, and x^2 is the output. When f(3) is written, 3 is put in place of x and the result is 9. A function in programming works the same way: it takes the input, computes, and gives the result.

Functions in programming differ from those in mathematics in two ways. First, some functions have no input. A function that prints a fixed greeting needs no data at all. Second, some functions return no value; their job is to leave an effect, such as writing to the screen. Java marks this case with the keyword `void` . In English “void” means “empty”: the method gives nothing back.

3.2.2 Defining and calling methods

There are two separate steps in using a method. First the method is **defined**: its name, inputs, result type and body are written down. A definition on its own runs nothing; it only records the recipe. Then the method is **called**: control moves to the method, the body runs, and control returns to the place where the call was made. A method is defined once but can be called as many times as needed.

3.2.2.1 Parts of a method header

A method definition has two parts: the **header** and the **body**. Figure 3.2.2 shows the parts of the `add` method, which adds two numbers, one by one.

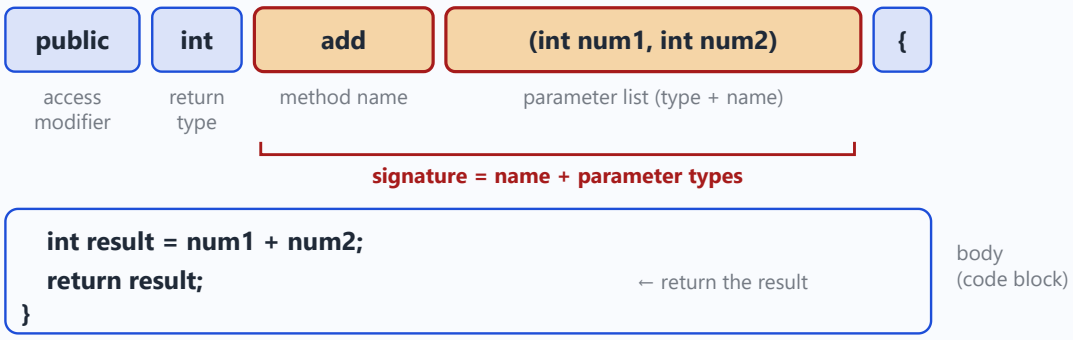


Figure 3.2.2: The parts of a method definition. The header contains, in order, the access modifier (`public`), the return type (`int`), the method name (`add`) and the parameter list. The name together with the parameter types forms the signature of the method. The part between the braces is the body.

Part	In the example	Role
Access modifier	`public`	Says who may call the method. `public` means it can be called from anywhere.
Return type	`int`	The type of the value the method gives back. A method that returns no value uses `void` .
Method name	`add`	The name used when the method is called. It is a verb or verb phrase and starts with a lowercase letter.
Parameter list	`(int num1, int num2)`	The inputs of the method. The type and name of each parameter are written out, and parameters are separated by commas.
Body	`{ ... }`	The statements that run when the method is called.

Table 3.2.1: The parts of the header `public int add(int num1, int num2) { ... }` .

Other **modifiers** can appear between the access modifier and the return type. The one seen most often in these notes is `static` , which says the method can be called without an object. The order in a header is fixed: modifiers first, then the return type, then the name, and finally the parameter list. The access modifier may be omitted; the method can then be called only from classes in the same package. The return type, however, can never be left out.

Java's naming convention is **camelCase**: the first word starts with a lowercase letter and the first letter of each following word is capitalized. `computeAverage` , `isPrime` and `readGrades` follow this rule. Because a method does something, its name states an action. For methods that return `boolean` , the name often reads like a question: `isPrime(17)` answers the question “is 17 prime?”.

Definition 3.2.2

The **method signature** is the name of the method together with the sequence of its parameter types. The signature of `add` is `add(int, int)`. The return type, the parameter names and the access modifier are not part of the signature.

The signature matters in two places. First, the compiler looks at the signature to decide which method a call refers to. Second, a class cannot contain two methods with the same signature. This rule decides when methods with the same name can be written side by side.

The compiler catches most mistakes made when writing a header. Three typical examples and the messages JDK 21 gives for them follow. In the first, the type of the second parameter has been forgotten. In mathematics, saying “let a, b be integers” is normal, but in Java every parameter carries its own type:

```
public class CommaParam {
    static int sum(int a, b) { return a + b; }
}
```

```
CommaParam.java:2: error: <identifier> expected
    static int sum(int a, b) { return a + b; }
                          ^
1 error
```

In the second, the return type is missing. The compiler accepts a header without a return type only for a **constructor**, whose name is the same as the class name. A constructor is a special method that runs when an object is created. Since `sum` is not the name of the class, an error is reported:

```
public class UntypedMethod {
    static sum(int a, int b) { return a + b; }
}
```

```
UntypedMethod.java:2: error: invalid method declaration; return type required
    static sum(int a, int b) { return a + b; }
           ^
1 error
```

In the third, the return type has been written before the modifier. The compiler expects a name after `int`, and when it meets `static` it reports two separate errors:

```
public class StaticOrder {
    int static sum(int a, int b) { return a + b; }
}
```

```
StaticOrder.java:2: error: <identifier> expected
    int static sum(int a, int b) { return a + b; }
       ^
StaticOrder.java:2: error: invalid method declaration; return type required
    int static sum(int a, int b) { return a + b; }
               ^
2 errors
```

Another rule is that methods cannot be nested. Every method is written in the body of the class, outside the other methods. If a method definition is placed inside `main`, the compiler sees it as a meaningless expression:

```
public class NestedCalls {
    public static void main(String[] args) {
        static int square(int n) { return n * n; }
    }
}
```

```
NestedCalls.java:3: error: illegal start of expression
        static int square(int n) { return n * n; }
        ^
NestedCalls.java:5: error: class, interface, enum, or record expected
}
^
2 errors
```

The order of methods in a class, on the other hand, does not matter. `main` can be written at the top with the helper methods below it, or the other way around. The compiler reads the whole class before it matches the calls.

3.2.2.2 Call, parameter and argument

To run a method you have defined, you write its name followed by its inputs in parentheses. The following program defines the `add` method and calls it from `main`.

```java
public class AddExample {
    public static int add(int num1, int num2) {
        int result = num1 + num2;
        return result;
    }

    public static void main(String[] args) {
        int result = add(5, 3);
        System.out.println("Sum: " + result);
    }
}
```

```
Sum: 8
```

Definition 3.2.3

A **method call** runs a method with arguments. A **parameter** is the variable in the method definition that stands for an input. An **argument** is the actual value given to the parameter at the time of the call.

In the program, the expression `add(5, 3)` is a call. The 5 and 3 in the call are arguments; `num1` and `num2` in the definition are parameters. An easy way to remember the difference: a parameter is written in the definition and has a name, while an argument is written in the call and has a value. At the moment of the call the values of the arguments are copied into the parameters in order: `num1 = 5`, `num2 = 3`.

When the call is made, control goes to the method. The body runs, `result` becomes 8, and `return result;` gives this value back. Control returns to the place where the call was made. The expression `add(5, 3)` now has the value 8, and this value is assigned to the variable `result` in `main`. The `println` on the last line is also a method call; its argument is the expression `"Sum: " + result`, and `Sum: 8` is printed.

Arguments must agree with the parameters in three respects: **number**, **order** and **type**. If a method with two parameters is given a single argument, the compiler rejects the call and prints the expected types next to the supplied ones:

```java
public class MissingArgument {
    static int add(int num1, int num2) { return num1 + num2; }
    public static void main(String[] args) {
        int s = add(5);
    }
}
```

```
MissingArgument.java:4: error: method add in class MissingArgument cannot be applied to
given types;
        int s = add(5);
                ^
  required: int,int
  found:    int
  reason: actual and formal argument lists differ in length
1 error
```

In the message, the line `required: int,int` shows the types the method expects, and `found: int` shows what the call supplied. The `reason` line says that the problem is the number of arguments. A type mismatch is also an error. In the call `add(5, 2.5)`, 2.5 is a `double`, which does not fit into an `int` parameter without losing information:

```
ArgumentType.java:4: error: incompatible types: possible lossy conversion from double to
int
        int s = add(5, 2.5);
                       ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full
output
1 error
```

The conversion in the opposite direction is allowed. An `int` argument can be given to a method with a `double` parameter; the value is automatically widened to `double`. This rule also determines the compiler's choice in overloading.

An argument does not have to be a constant number; it can be a variable or any expression. In the call `square(a + 1)`, Java first evaluates the expression `a + 1`. If `a` is 3, the value of the argument is 4, and this 4 is copied into the parameter. The method sees neither the variable `a` nor the expression `a + 1`; it receives only the value 4. This rule is the basis of the topic of copying arguments.

3.2.2.3 Tracing a call step by step

To understand what happens in memory when a method is called, two concepts are needed: the stack frame and the call stack.

Definition 3.2.4

A **stack frame** is the memory record that holds the parameters and local variables of one method call. The **call stack** is the structure that keeps the frames of the running methods on top of one another. Each call pushes a new frame onto the top of the stack; when the method finishes, its frame is removed from the stack.

A stack is like a pile of plates. The last plate put down is on top and is the first one taken. On the call stack, too, the frame of the most recently called method is on top. The running method always owns the topmost frame. When that method finishes, its frame is removed and the method below it continues where it left off.

The execution of the previous program is given step by step in Table 3.2.2. The line numbers follow the code layout of the trace slide in the lecture slides: lines 1–4 are the `add` method and lines 6–9 are the `main` method.

Step	Line	Call stack	Variables	Output
1	6: `public static void main(String[] args) {`	main()	args=[]	—
2	7: `int result = add(5, 3);`	main() → add(5, 3)	num1=5, num2=3	—
3	2: `int result = num1 + num2;`	main() → add(5, 3)	num1=5, num2=3, result=8	—
4	3: `return result;`	main() → add(5, 3)	num1=5, num2=3, result=8, return=8	—
5	7: `int result = add(5, 3);`	main()	args=[], result=8	—
6	8: `System.out.println("Sum: " + result);`	main()	args=[], result=8	`Sum: 8`

Table 3.2.2: Tracing the call `add(5, 3)`. In the call stack column the rightmost frame is on top; the variables column shows the variables of that frame.

In the first step the stack contains only the frame of `main`. A program always starts at the `main` method; `args` is the array that holds the command-line arguments, and here it is empty. In the second step `add(5, 3)` is called. A new frame is opened on top of the stack, and the arguments are copied into the parameters in this frame.

In the third step the body runs: `result = 5 + 3 = 8`. This `result` is a local variable of `add` and lives in the frame of `add`. Figure 3.2.3 shows this moment. The frame of `main` also has a variable named `result`, but its value is not yet known. The two variables have the same name, yet they are separate variables living in different frames.

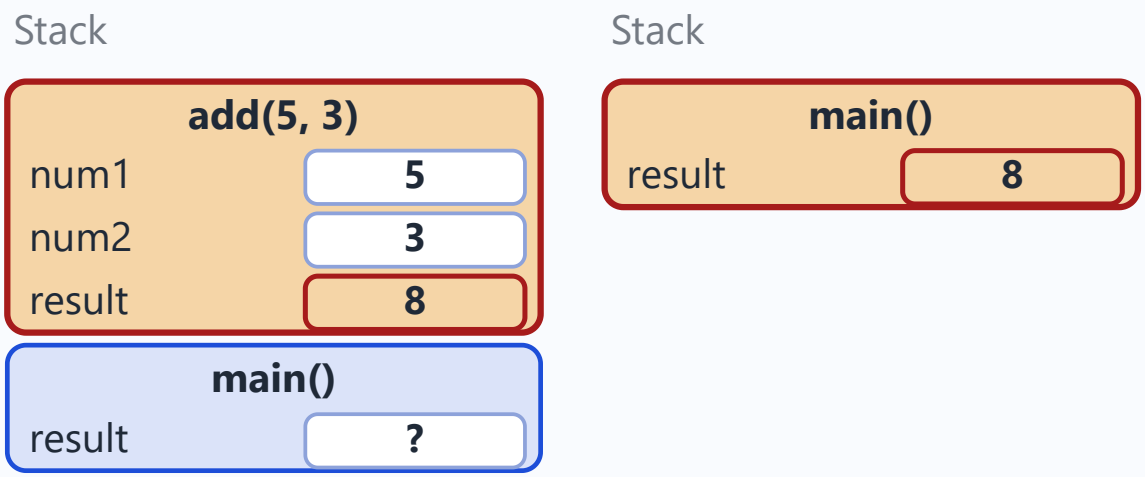


Figure 3.2.3: Two moments of the call stack. On the left `add(5, 3)` is running: the top frame holds `num1 = 5`, `num2 = 3`, `result = 8`; in the `main` frame below, `result` is still undefined. On the right the method has returned and its frame is gone; `result` in `main` is now 8.

In the fourth step `return result;` runs. The value 8 is sent to the caller, the method ends, and its frame is removed from the stack. Together with the removed frame, `num1`, `num2` and the `result` of `add` disappear. In the fifth step control returns to line 7, where the call was made; the returned 8 is assigned to the variable `result` of `main`. In the sixth step `println` is called and `Sum: 8` is printed. `println` is a method too; for the short time it runs, a frame for it is pushed onto the stack and popped again.

The lesson of this trace is that a method call is control temporarily going somewhere else and coming back. A new workspace is opened at the destination, and when the job is done this workspace is cleaned up. Only the value sent with `return` remains.

3.2.2.4 Methods that return a value and methods that do not

The following program contains three methods that differ in their input and output.

```
public class ExampleMethods {
    // Addition method
    public static int add(int num1, int num2) {
        return num1 + num2;
    }
    // Squaring method
    public static double square(double num) {
        return num * num;
    }
    // Hello world method
    public static void helloWorld() {
        System.out.println("Hello, World!");
    }

    public static void main(String[] args) {
        System.out.println(add(2, 7));
        System.out.println(square(1.5));
        helloWorld();
    }
}
```

```
9
2.25
Hello, World!
```

Method	Input	Output	Use of the call
`add`	2 × `int`	`int`	`println(add(2, 7))` → 9
`square`	1 × `double`	`double`	`println(square(1.5))` → 2.25
`helloWorld`	none	none (`void`)	`helloWorld();` as a statement on its own

Table 3.2.3: Inputs and outputs of the three methods.

This time `add` returns the result directly without an intermediate variable: `return num1 + num2;`. Java first evaluates the expression and then returns the value. The two ways of writing it, with and without the intermediate variable, give the same result.

The call `square(1.5)` returns 1.5 × 1.5 = 2.25, and `2.25` is printed. Java writes decimal numbers with a point, both in code and in `println` output. The method `helloWorld` has neither a parameter nor a return value. The parentheses are still written: `helloWorld()` in the definition and `helloWorld();` in the call. Writing `helloWorld;` without parentheses is a compile error, because Java does not recognize a name as a method call unless it sees parentheses.

The real difference between the two kinds of method is in how the call is used. A call to a method that returns a value is an **expression**; it has a value. This value can be assigned to a variable, used in another calculation, or passed as an argument to another method: in `println(add(2, 7))` the result

of `add` becomes the argument of `println`. A call to a `void` method has no value. It can be written only as a **statement** on its own.

A method that returns a value can also be called on its own: `add(2, 7);` is a valid statement. But because the returned value is not assigned anywhere, it is lost. The compiler gives no warning in this case. That is why, in most complaints of the form “I called the method but the result did not change”, the returned value was never assigned to a variable.

3.2.3 return and void

The `return` statement gives the result of a method to the caller. Its rules are simple, but it is one of the places where beginners argue most with the compiler. This section explains how `return` works, when the type of the returned value is accepted, and how the compiler checks the rule “every path must end in a `return`”.

3.2.3.1 The return statement

Definition 3.2.5

The **return value** is the value the method gives to the caller with `return`. The **return type** is the type of this value; it is written in the header before the method name. If the method returns no value, the return type is `void`.

The statement `return expression;` has two effects. First, the value of the expression is computed and handed to the caller. Second, the method ends **immediately**. Lines after `return` do not run. This property is used to leave a method early.

A `void` method returns no value, but it can still end early. For that, a bare `return;` is written. The following program checks the divisor before dividing. If the divisor is zero, it prints a warning and leaves the method at once.

```
public class EarlyReturn {
    static void printQuotient(int dividend, int divisor) {
        if (divisor == 0) {
            System.out.println("Cannot divide by zero!");
            return;                     // leave the method at once
        }
        System.out.println(dividend / divisor);
    }
    public static void main(String[] args) {
        printQuotient(10, 2);
        printQuotient(7, 0);
    }
}
```

```
5
Cannot divide by zero!
```

In the first call `divisor` is 2. The `if` condition is false, its body is skipped, and the result of `10 / 2`, which is 5, is printed. In the second call `divisor` is 0. The condition is true; the warning is printed and `return;` ends the method. The line `println(dividend / divisor)` is never reached. Had it been reached, the integer division would have tried to divide by zero and the program would have stopped with `ArithmeticException: / by zero`.

The same method could have been written with `else`. The advantage of the early-`return` pattern is that it filters out the bad cases at the start of the method and leaves the real work unindented. As the number of conditions to check grows, this style is more readable than nested `if`s.

Writing a statement right after `return` in the same block is a compile error. The compiler sees that this statement cannot be reached by any path and reports `unreachable statement`:

```
public class Unreachable {
    static int f(int n) {
        return n * 2;
        System.out.println("done");
    }
}
```

```
Unreachable.java:4: error: unreachable statement
        System.out.println("done");
        ^
1 error
```

3.2.3.2 Matching the return type

The type of the value that `return` returns must be compatible with the return type declared in the header. The compatibility rule is the same as for assignment: a value can be returned if it can be assigned to a variable of the return type. So an `int` value can be returned from a method with return type `double`; the value is widened automatically. The opposite means a loss of information, and the compiler does not accept it:

```
public class Narrowing {
    static int half(double x) { return x / 2; }
}
```

```
Narrowing.java:2: error: incompatible types: possible lossy conversion from double to int
    static int half(double x) { return x / 2; }
                                       ^
1 error
```

Because widening is free, a trap arises that compiles but gives a wrong result. Both of the following methods return `double`, but their results differ.

```
public class Widening {
    // the int result is widened to the double return type
    static double half(int n) {
        return n / 2;
    }
    static double halfFixed(int n) {
        return n / 2.0;
    }
    public static void main(String[] args) {
        System.out.println(half(7));
        System.out.println(halfFixed(7));
    }
}
```

```
3.0
3.5
```

In `half(7)`, `n / 2` is a division between two `int`s. Integer division discards the fraction and the result is 3. Making the return type `double` does not change this calculation; the value 3 is widened to 3.0 as it is returned. In other words, the return type decides not how the expression is evaluated, but which type the computed result is converted to. For the correct result the division itself must be made floating-point: the expression `n / 2.0` gives 3.5.

`void` and returning a value exclude each other. A value cannot be returned with `return` from a `void` method. Nor can a bare `return;` be written in a method that returns a value. The compiler gives separate messages for the two cases:

```
public class VoidValue {
    static void greet() { return "Hello"; }
}
```

```
VoidValue.java:2: error: incompatible types: unexpected return value
    static void greet() { return "Hello"; }
                                 ^
1 error
```

```
public class IntBareReturn {
    static int twice(int n) { int s = n * 2; return; }
}
```

```
IntBareReturn.java:2: error: incompatible types: missing return value
    static int twice(int n) { int s = n * 2; return; }
                                               ^
1 error
```

Trying to use the result of a `void` method is also an error. Since a call to a `void` method has no value, it can be neither assigned to a variable nor passed as an argument to `println`:

```
public class VoidUse {
    static void sayHello() { System.out.println("Hi"); }
    public static void main(String[] args) {
        int x = sayHello();
    }
}
```

```
VoidUse.java:4: error: incompatible types: void cannot be converted to int
        int x = sayHello();
                ^
1 error
```

```
VoidPrint.java:4: error: 'void' type not allowed here
        System.out.println(sayHello());
                                   ^
1 error
```

3.2.3.3 Every path must end in a return

In a method that returns a value, every path that control can follow must end in a `return`. If control can reach the closing brace at the end of the method without passing through any `return`, the compiler says `missing return statement`. The most surprising form of the error appears in chains of conditions that look complete to a human:

```
public class MissingReturn {
    static int sign(int n) {
        if (n > 0) return 1;
        if (n <= 0) return -1;
    }
}
```

```
MissingReturn.java:5: error: missing return statement
    }
    ^
1 error
```

To a human, the conditions `n > 0` and `n <= 0` cover all integers; one of the two is certainly true. The compiler, however, does not reason about the values of conditions. It looks only at the structure: there are two `if`s, and both are treated as if they could be false. Control can then reach the end of the method, and there is no `return` there. The same error appears with an `if` / `else if` chain; as long as the last branch is an `else if`, the compiler assumes the chain can be left somewhere.

The fix is to turn the last condition into a plain `else`, or to put a default `return` at the end of the method:

```
static int sign(int n) {
    if (n > 0) return 1;
    return -1;          // every case with n <= 0 ends up here
}
```

`void` methods have no such requirement. When control reaches the end of the method, the method ends by itself; writing `return;` at the end is unnecessary.

3.2.4 static methods and calls through an object

Every method so far had the word `static` in its header. This word determines how the method is called. Without going into the details of objects, it is enough for now to tell the two forms of call apart.

Definition 3.2.6

A **static method** belongs to the class, not to an object. No object is needed to call it. It is called by its name from inside the same class, and from another class in the form `ClassName.method()`. A method that is not `static` is called through an **object**: `object.method()`.

The most familiar example of a `static` method is the `Math` class. To take a square root, no `Math` object is created; you simply write `Math.sqrt(16)`. Inside the same class there is no need to write the class name:

```
static int add(int num1, int num2) {
    return num1 + num2;
}
// inside main, directly:
int result = add(5, 3);
```

To call a method that is not `static`, you first create an object of that class with `new`, and then call the method through the object:

```
public class Calculator {
    int add(int num1, int num2) {
        return num1 + num2;
    }
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int result = calc.add(5, 3);
        System.out.println(result);  // 8
    }
}
```

```
8
```

The line `Calculator calc = new Calculator();` creates an object of the `Calculator` class and reaches it through a variable named `calc`. The call `calc.add(5, 3)` runs the method on this object. The dot means “the method of this object”. The result is again 8.

If the two forms are mixed up, the compiler reports an error. `main` is a `static` method; while it runs, no object exists yet. Calling a non-`static` method directly from `main` leaves the question "whose method?" unanswered:

```
public class StaticContext {
    int add(int num1, int num2) { return num1 + num2; }
    public static void main(String[] args) {
        int result = add(5, 3);
    }
}
```

```
StaticContext.java:4: error: non-static method add(int,int) cannot be referenced from a
static context
        int result = add(5, 3);
                     ^
1 error
```

"Static context" in the message means the inside of a `static` method. There are two ways to fix it: either add `static` to the header of the method, or create an object and call the method through it. Because the methods in this chapter do not need the state of an object, all of them are written `static`.

This also explains why `main` itself is `static`. When the program starts, the JVM (Java Virtual Machine) has not created any object yet. For it to call `main` without needing an object, the method must belong to the class.

3.2.5 Pass by value

Can a method change a variable that is given to it as an argument? The answer is one of the most misunderstood points in Java. The short answer is no. The method receives a copy of the argument, not the argument itself. But for arrays and objects the thing that is copied is a reference, so the situation looks different at first sight. This section follows the two cases separately.

Definition 3.2.7

Pass by value means that at the time of the call the value of the argument is copied and given to the parameter. Java passes all arguments by value. If the argument is an array or an object, the value that is copied is the reference that points to the location of that array or object.

3.2.5.1 A primitive argument

In the following program both `main` and `increase` have a variable named `num`. `increase` adds 10 to its own `num`.

```
static void increase(int num) {
  num = num + 10;
  System.out.println("increase: " + num);
}

public static void main(String[] args) {
  int num = 5;
  increase(num);
  System.out.println("main: " + num);
}
```

```
increase: 15
main: 5
```

The program prints `increase: 15` and `main: 5`. The `num` in `main` has not changed. Table 3.2.4 shows why, step by step.

Step	Line	Call stack	Variables	Output
1	7: `int num = 5;`	main()	num=5	—
2	8: `increase(num);`	main() → increase(5)	num=5	—
3	2: `num = num + 10;`	main() → increase(5)	num=15	—
4	3: `System.out.println("increase: " + num);`	main() → increase(5)	num=15	`increase: 15`
5	9: `System.out.println("main: " + num);`	main()	num=5	`main: 5`

Table 3.2.4: Tracing the call `increase(num)`. The variables column shows the topmost frame.

In the second step `increase(num)` is called. Java reads the value of the argument variable `num`: 5. In the newly opened frame a separate box is set aside for the parameter of `increase`, and 5 is copied into this box. There are now two separate `num`s in memory: one in the frame of `main` and one in the frame of `increase`. It does not matter that their names are the same; Java keeps the variables of each frame apart.

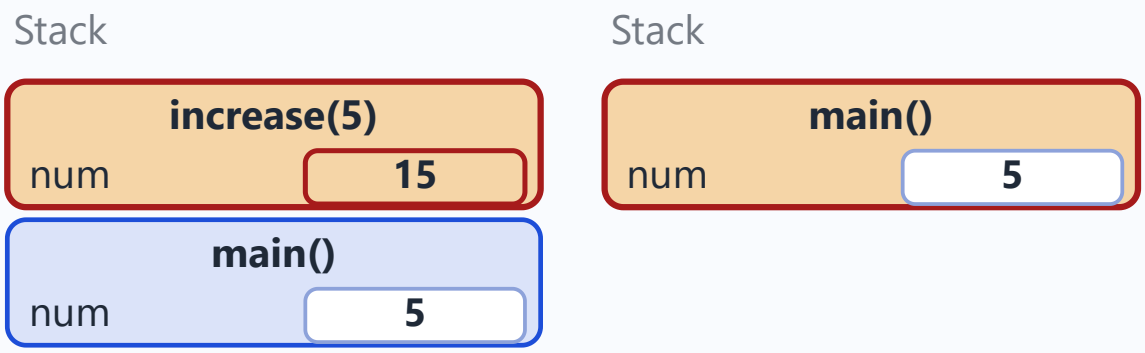


Figure 3.2.4: Pass by value. On the left `increase` is running: in the top frame the copy has become 15, while in the `main` frame below `num` is still 5. On the right the method has finished; the copy was removed along with the frame, and the 5 in `main` is untouched.

In the third step `num = num + 10;` runs. Because this line runs in the topmost frame, that is, inside `increase`, it changes only the copy: the copy becomes 15. In the fourth step `increase: 15` is printed. In the fifth step the method ends; its frame, and with it the copy, is removed. When control returns to `main`, the `num` of `main` is 5, exactly as untouched as before.

The only way to carry the method's change back to `main` is `return`. `increase` is written so that it returns the new value, and the caller assigns this value to its own variable:

```
static int increase(int num) {
    return num + 10;
}
// inside main:
num = increase(num);   // num in main is now 15
```

Here too the method does not touch the variable of `main`. The one that makes the change is `main` itself, which writes the assignment. This arrangement makes programs safe: when a method is called, it is certain that the local variables of the caller will not change.

3.2.5.2 An array argument: a copy of the reference

With methods that take an array parameter the situation looks different. First you need to know where an array lives in memory. In Java, memory is divided into two regions. Frames and local variables live on the **stack**. Arrays and objects are kept in a separate region called the **heap**. An array variable holds not the array itself but a **reference** to the location of the array on the heap.

The following method takes an array and an `int`. It tries to change the array and the integer, and then assigns a brand new array to its parameter.

```
static void modify(int[] d, int x) {
    d[0] = 50;
    x = 50;
    d = new int[2];
    d[1] = 50;
}

public static void main(String[] args) {
    int[] a = {1, 2};
    int x = 1;
    modify(a, x);
    System.out.println(a[0] + " " + a[1] + " " + x);
}
```

```
50 2 1
```

The program prints `50 2 1`: the first element of the array has changed, the second has not, and `x` has not changed at all. Table 3.2.5 shows the state of the two arrays and the two `x` variables at each step. In the table, the array created in `main` is called A and the new array created inside the method is called B.

Step	Line	`a` → array	`d` → array	`x` (main)	`x` (modify)
1	9: `int[] a = {1, 2};`	A {1, 2}	—	—	—
2	10: `int x = 1;`	A {1, 2}	—	1	—
3	1: `static void modify(int[] d, int x) {`	A {1, 2}	A {1, 2}	1	1
4	2: `d[0] = 50;`	A {50, 2}	A {50, 2}	1	1
5	3: `x = 50;`	A {50, 2}	A {50, 2}	1	50
6	4: `d = new int[2];`	A {50, 2}	B {0, 0}	1	50
7	5: `d[1] = 50;`	A {50, 2}	B {0, 50}	1	50
8	11: `modify(a, x);`	A {50, 2}	—	1	—
9	12: `System.out.println(a[0] + " " + a[1] + " " + x);`	A {50, 2}	—	1	—

Table 3.2.5: Tracing the call `modify(a, x)`. The notation "A {50, 2}" says that the variable points to array A and gives the current contents of the array.

In the first step `int[] a = {1, 2};` creates an array of two elements on the heap; `a` holds the reference to this array. In the second step `x = 1`. Because it is a primitive value, the 1 is stored directly in the frame of `main`.

In the third step `modify(a, x)` is called and both arguments are passed by value. `d` is given a copy of the reference in `a`; the `x` of `modify` is given a copy of 1. The reference has been copied, but the array has not. `a` and `d` now point to the same array A on the heap. The first part of Figure 3.2.5 draws this moment: two arrows in two separate frames reach the same array.

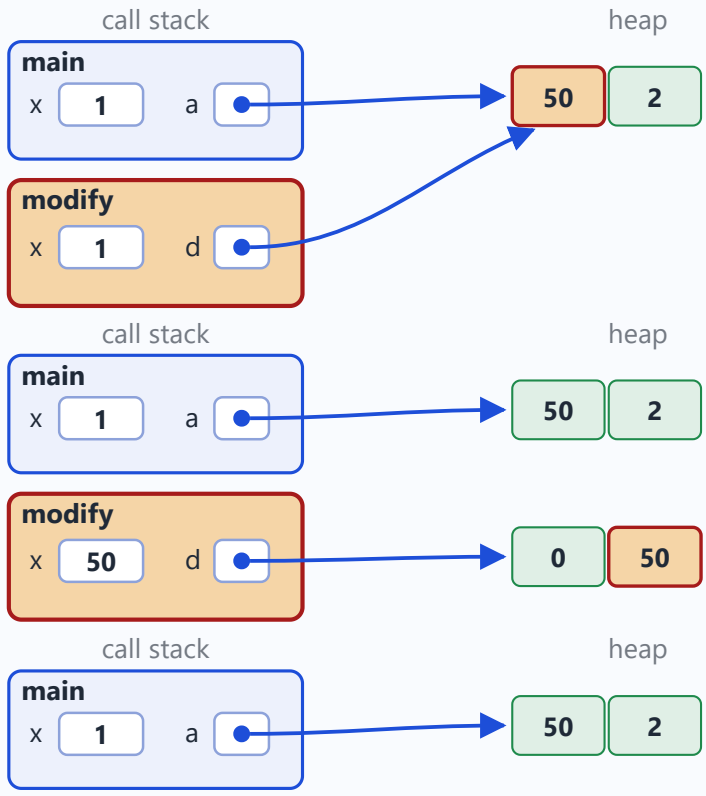


Figure 3.2.5: A copy of the reference. Top: after `d[0] = 50`; `a` and `d` point to the same array, and the change is written to the shared array. Middle: after `d = new int[2]` and `d[1] = 50`; `d` now points to the new array, and the link is broken. Bottom: the method has ended; no reference to the new array is left.

In the fourth step `d[0] = 50;` runs. This assignment changes not the variable `d` but the first cell of the array that `d` points to. Because `d` and `a` point to the same array, the change is visible from `main` too. In the fifth step `x = 50;` changes only the copy in `modify`; the `x` in `main` stays 1.

In the sixth step `d = new int[2];` creates a new array B on the heap and assigns the reference to this array to `d`. This assignment changes `d` itself. Because `d` is a copy, `a` is not affected; `a` still points to A. The link between `d` and `a` is broken on this line. The `d[1] = 50;` in the seventh step writes to the new array; array A does not change.

In the eighth step the method ends and its frame is removed. The only reference to array B was `d`; now no variable points to B. Such an array can never be reached from the program again. The JVM's **garbage collector** reclaims the memory of such objects when the time comes. In the ninth step `a[0]`, `a[1]` and `x` are printed: `50 2 1`.

The rule that follows from this example is the following. Java passes everything by value; a reference is also a value that is copied. A method can change the array that the reference it received points to, because the array is shared. But assigning a new array to the parameter does not change the caller's variable, because the parameter is only a copy. Some sources write that Java passes objects "by reference". This statement is wrong: with true pass by reference, the line `d = new int[2]` would have changed `a` in `main` as well.

3.2.5.3 The swap trap

The best-known consequence of pass by value is that a method that exchanges the values of two variables cannot be written in Java. In the following program there are two versions of the `swap` method. The first takes two `int`s; the second takes an array and two indices.

```
import java.util.Arrays;

public class Swap {
    static void swap(int x, int y) {
        int tmp = x;
        x = y;
        y = tmp;
    }
    static void swap(int[] d, int i, int j) {
        int tmp = d[i];
        d[i] = d[j];
        d[j] = tmp;
    }
    public static void main(String[] args) {
        int[] arr = {4, 7, 9};
        swap(arr[0], arr[2]);
        System.out.println(Arrays.toString(arr));
        swap(arr, 0, 2);
        System.out.println(Arrays.toString(arr));
    }
}
```

```
[4, 7, 9]
[9, 7, 4]
```

The first call is `swap(arr[0], arr[2])`. The arguments are the values of the expressions `arr[0]` and `arr[2]`, that is, 4 and 9. The method swaps these two copies inside itself: `x` becomes 9 and `y` becomes 4. When the method ends, the copies are removed and the array stays unchanged: `[4, 7, 9]`.

The second call takes the array and the indices. This time the method writes to the cells of the array that the reference points to: `d[0]` becomes 9 and `d[2]` becomes 4. Because the array is shared, the change is visible from `main`: `[9, 7, 4]`. `Arrays.toString` is a built-in method that prints an array in square brackets.

The behavior of pass by value can also be followed step by step in the Pass-by-Value Lab simulator. The simulator draws the passing of a primitive value, an array, and an object separately on the call stack and the heap.

3.2.6 Method overloading

Sometimes the same job has to be done with inputs of different types or in different numbers. Consider methods that add two integers, two floating-point numbers and three integers. It is possible to give each a separate name: `sumTwoInts`, `sumTwoDoubles`, `sumThreeInts`. But such names make use harder; the programmer has to remember the right name every time. Java allows the same name to be used for several methods.

3.2.6.1 Same name, different parameter list

Definition 3.2.8

Method overloading means defining several methods with the same name in the same class. The parameter lists of the methods must differ: in the number, the types or the order of the parameters. In other words, overloaded methods have different signatures.

```
public class Overloading {
    static int sum(int a, int b) {
        return a + b;
    }
    static double sum(double a, double b) {
        return a + b;
    }
    static int sum(int a, int b, int c) {
        return a + b + c;
    }

    public static void main(String[] args) {
        System.out.println(sum(2, 3));
        System.out.println(sum(2.5, 1.2));
        System.out.println(sum(1, 2, 3));
    }
}
```

```
5
3.7
6
```

The class has three `sum` methods. Their signatures are `sum(int, int)`, `sum(double, double)` and `sum(int, int, int)`; all three differ from each other. The call `sum(2, 3)` carries two `int` arguments and runs the first method: 5. `sum(2.5, 1.2)` carries two `double`s and runs the second: 3.7. `sum(1, 2, 3)` carries three arguments and runs the third: 6. The different appearance of `5` and `3.7` in the output also gives away the method chosen: the result of the method that returns `int` is printed without a decimal point.

It is the **compiler** that decides which method will run. The decision is made before the program runs, at compile time, and only by looking at the types of the arguments. In the compiled program every call is already bound to a specific method.

The Java library uses overloading heavily. `println` is not a single method: `println(int)`, `println(double)`, `println(String)`, `println(char)`, `println(boolean)` and the parameterless `println()` are defined separately. The programmer sees them all as a single name and uses the same name whatever type is printed. `Math.abs` and `Math.max` are also overloaded separately for `int`, `long`, `float` and `double`.

The criterion for overloading is the signature, and the return type is not part of the signature. For this reason two methods that differ only in their return type cannot exist in the same class:

```
public class ReturnType {
    static int f(int a) { return a; }
    static double f(int a) { return a; }
}
```

```
ReturnType.java:3: error: method f(int) is already defined in class ReturnType
    static double f(int a) { return a; }
                  ^
1 error
```

The reason for the rule is clear. In a call such as `f(5);` that does not use the returned value, the compiler could not tell which of the two methods to choose. Different parameter names are not enough either: `sum(int a, int b)` and `sum(int x, int y)` have the same signature.

3.2.6.2 Which version does the compiler choose?

When the types of the arguments are exactly the same as the parameters, the choice is easy. The hard part is what happens when there is no exact match. In that case the compiler looks for methods that fit by widening the arguments. The widening conversions in Java follow this order:

> `byte → short → int → long → float → double` and `char → int`

A type can automatically widen to every type that comes after it in this chain. Narrowing conversions in the opposite direction are never made automatically. The choice can be thought of in three steps:

1. Methods whose number of parameters differs from the number of arguments are eliminated.
2. Of the remaining methods, those in which each argument either has the same type or fits by widening are considered **applicable**.
3. If several methods are applicable, the compiler chooses the **most specific** one. A method is more specific if all its parameter types can widen to the parameter types of the other. The `(int, int)` version is more specific than `(double, double)`, because `int` widens to `double`.

The following program tries the same three methods with four different calls. Its output and trace follow.

```
static int sum(int a, int b) {
    return a + b;
}
static double sum(double a, double b) {
    return a + b;
}
static int sum(int a, int b, int c) {
    return a + b + c;
}
public static void main(String[] args) {
    System.out.println(sum(4, 6));
    System.out.println(sum(4, 2.5));
    System.out.println(sum(7, 1, 2));
    System.out.println(sum('A', 1));
}
```

```
10
6.5
10
66
```

Step	Line	Call stack	Variables	Output
1	11: `System.out.println(sum(4, 6));`	main()	—	—
2	2: `return a + b;`	main() → sum(int, int)	a=4, b=6	—
3	11: `System.out.println(sum(4, 6));`	main()	—	`10`
4	12: `System.out.println(sum(4, 2.5));`	main()	—	—
5	5: `return a + b;`	main() → sum(double, double)	a=4.0, b=2.5	—
6	12: `System.out.println(sum(4, 2.5));`	main()	—	`6.5`
7	13: `System.out.println(sum(7, 1, 2));`	main()	—	—
8	8: `return a + b + c;`	main() → sum(int, int, int)	a=7, b=1, c=2	—
9	13: `System.out.println(sum(7, 1, 2));`	main()	—	`10`
10	14: `System.out.println(sum('A', 1));`	main()	—	—
11	2: `return a + b;`	main() → sum(int, int)	a=65, b=1	—
12	14: `System.out.println(sum('A', 1));`	main()	—	`66`

Table 3.2.6: The version of `sum` chosen in four calls. The call stack column shows the signature of the chosen version.

The **first call** is `sum(4, 6)`; the arguments are `(int, int)`. The version with three parameters is eliminated because of the count. `(int, int)` is an exact match; `(double, double)` also fits by widening. The most specific, `(int, int)`, is chosen and 10 is printed. The first part of Figure 3.2.6 shows this choice.

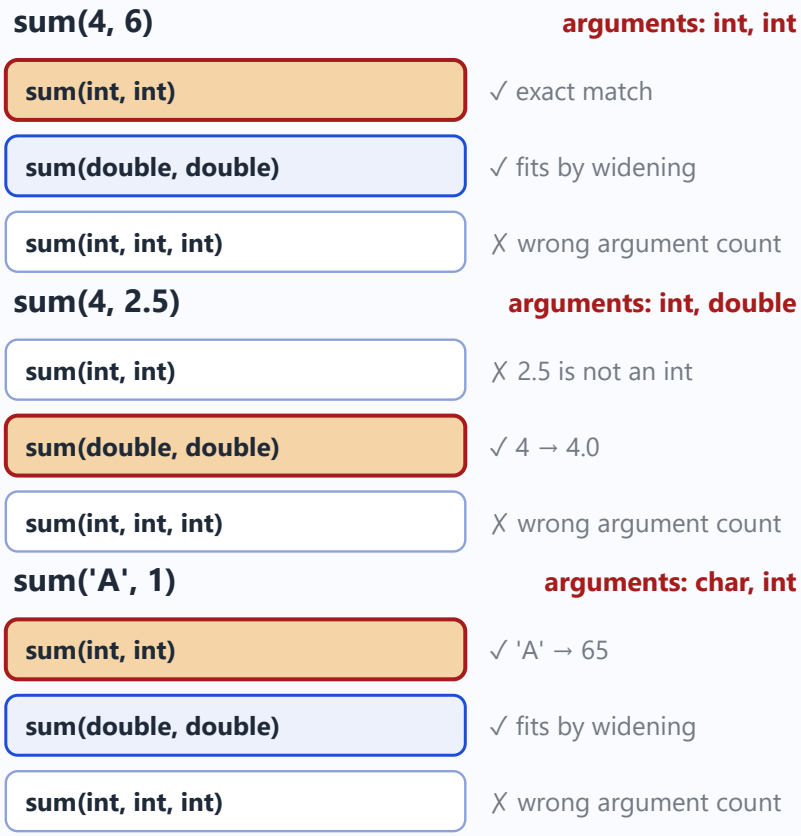


Figure 3.2.6: How the compiler evaluates the candidate methods. Top, `sum(4, 6)`: `(int, int)` is an exact match. Middle, `sum(4, 2.5)`: 2.5 cannot be narrowed to `int`, so `(int, int)` is eliminated; 4 is widened and `(double, double)` is chosen. Bottom, `sum('A', 1)`: `char` can widen to both versions; the most specific, `(int, int)`, is chosen.

The **second call** is `sum(4, 2.5)`; the arguments are `(int, double)`. The `(int, int)` version does not fit, because 2.5 cannot be converted implicitly to `int`. `(double, double)` fits: the value 4 is widened to 4.0. The method runs with `a = 4.0`, `b = 2.5` and returns 6.5.

The **third call** `sum(7, 1, 2)` carries three arguments. Only the version with three parameters fits, and it returns 10. In the output the first and the third lines are both 10, but they come from different methods.

The **fourth call** is `sum('A', 1)`; the arguments are `(char, int)`. In Java the type `char` holds the numeric code of a character and can widen to both `int` and `double`. Both versions are applicable; the compiler chooses the most specific, `(int, int)`. Since the code of the character `'A'` is 65, the method runs with `a = 65`, `b = 1` and returns 66. What appears on the screen is not a letter but the number 66.

The consequences of the widening chain are sometimes surprising. In the following program the `print` method has `int`, `long` and `double` versions; each version prints a different letter.

```
public class Print {
    static void print(int a) {
        System.out.print("I");
    }
    static void print(long a) {
        System.out.print("L");
    }
    static void print(double a) {
        System.out.print("D");
    }
    public static void main(String[] args) {
        print(5); print(5L);
        print(5.0f); print('A');
        System.out.println();
        byte b = 3; short s = 4;
        print(b); print(s); print(5_000_000_000L);
        System.out.println();
    }
}
```

```
ILDI
IIL
```

`print(5)` is an exact match: I. The suffix L in `print(5L)` makes the number a `long`: L. The suffix f in `print(5.0f)` makes the number a `float`. There is no `float` version; in the chain `float` can widen only to `double`: D. In `print('A')`, `char` can widen to `int`, `long` and `double`; the most specific is `int`: I. On the second line the `byte` and `short` arguments go to the nearest version, `int`: I I. `5_000_000_000L` is a `long` beyond the range of `int`: L. The underscores in the number are only for readability; they do not change the value.

There are also cases in which no most specific method exists. If neither of two applicable methods is more specific than the other, the compiler makes no choice and rejects the call:

```
public class Ambiguous {
    static void g(int a, double b) { }
    static void g(double a, int b) { }
    public static void main(String[] args) {
        g(1, 1);
    }
}
```

```
Ambiguous.java:5: error: reference to g is ambiguous
        g(1, 1);
        ^
  both method g(int,double) in Ambiguous and method g(double,int) in Ambiguous match
1 error
```

In the call `g(1, 1)` both versions are applicable: in the first the second argument is widened to `double`, in the second the first argument is. Between `(int, double)` and `(double, int)` neither can widen to the other; neither is "more specific". The call is ambiguous, as the message says. If the call is written as `g(1, 1.0)`, the ambiguity disappears. This error shows why merely changing the order of the parameter types is not a good idea in overloading.

3.2.7 Methods calling methods

Methods are not called only from `main`; a method can also call another method. This is how a large task is split into small pieces. Each piece does its own job and hands its result to a higher level.

3.2.7.1 Methods that call each other

In the following program, the `sumOfSquares` method, which finds the sum of the squares of two numbers, does not do the work itself. It has `square` do the squaring and `sum` do the adding.

```java
public class Nested {
    public static int square(int num) {
        return num * num;
    }
    public static int sum(int num1, int num2) {
        return num1 + num2;
    }
    public static int sumOfSquares(int num1, int num2) {
        return sum(square(num1), square(num2));
    }
    public static void main(String[] args) {
        int result = sumOfSquares(3, 4);
        System.out.println(result);
    }
}
```

```
25
```

The result of the call `sumOfSquares(3, 4)` is $3^2 + 4^2 = 9 + 16 = 25$. The key line of the code is `return sum(square(num1), square(num2));`. In this line the arguments of one call are other calls. Java evaluates such an expression from the inside out. Before `sum` can be called, the values of its two arguments must be known; so `square(num1)` runs first, then `square(num2)`. When both results are ready, `sum(9, 16)` is called.

The order in which arguments are evaluated is precisely defined in Java: **left to right**. This rule matters when one of the arguments prints something or changes a variable. In the following program the `value` method both returns a value and prints its own name, so the order of the calls shows up in the output.

```
public class Order {
    static int value(String name, int v) {
        System.out.print(name + " ");
        return v;
    }
    static int diff(int a, int b) {
        return a - b;
    }
    public static void main(String[] args) {
        int s = diff(value("left", 10), value("right", 3));
        System.out.println("→ " + s);
    }
}
```

```
left right → 7
```

First `left` and then `right` is printed. Then `diff(10, 3)` runs and returns 7. Some languages, such as C, do not specify this order and leave it to the compiler. Java guarantees the order.

Having methods call each other makes the program readable and manageable. The body of `sumOfSquares` says in one line what it does: the sum of two squares. `square` and `sum` can be tested separately. If `square` works wrongly, the bug is fixed in one place and all the methods that use it are fixed.

3.2.7.2 Tracing the call stack

With nested calls the call stack has several levels. Table 3.2.7 gives the execution of the program step by step.

Step	Line	Call stack	Variables	Output
1	12: `int result = sumOfSquares(3, 4);`	main()	result=?	—

Step	Line	Call stack	Variables	Output
2	9: `return sum(square(num1), square(num2));`	main() → sumOfSquares(3, 4)	num1=3, num2=4	—
3	2: `return num * num;`	main() → sumOfSquares(3, 4) → square(3)	num=3, return=9	—
4	9: `return sum(square(num1), square(num2));`	main() → sumOfSquares(3, 4)	num1=3, num2=4, arg1=9	—
5	2: `return num * num;`	main() → sumOfSquares(3, 4) → square(4)	num=4, return=16	—
6	5: `return num1 + num2;`	main() → sumOfSquares(3, 4) → sum(9, 16)	num1=9, num2=16, return=25	—
7	9: `return sum(square(num1), square(num2));`	main() → sumOfSquares(3, 4)	num1=3, num2=4, return=25	—
8	12: `int result = sumOfSquares(3, 4);`	main()	result=25	—
9	13: `System.out.println(result);`	main()	result=25	`25`

Table 3.2.7: The call stack during the call `sumOfSquares(3, 4)`. The variables column shows the topmost frame; `arg1` is the computed first argument of `sum`.

In the first step `main` calls `sumOfSquares(3, 4)`; there are two frames on the stack. In the second step `sumOfSquares` starts evaluating its arguments before calling `sum`. In the third step a third frame is opened for `square(3)`; 3 × 3 = 9 is returned. The first part of Figure 3.2.7 shows this moment.

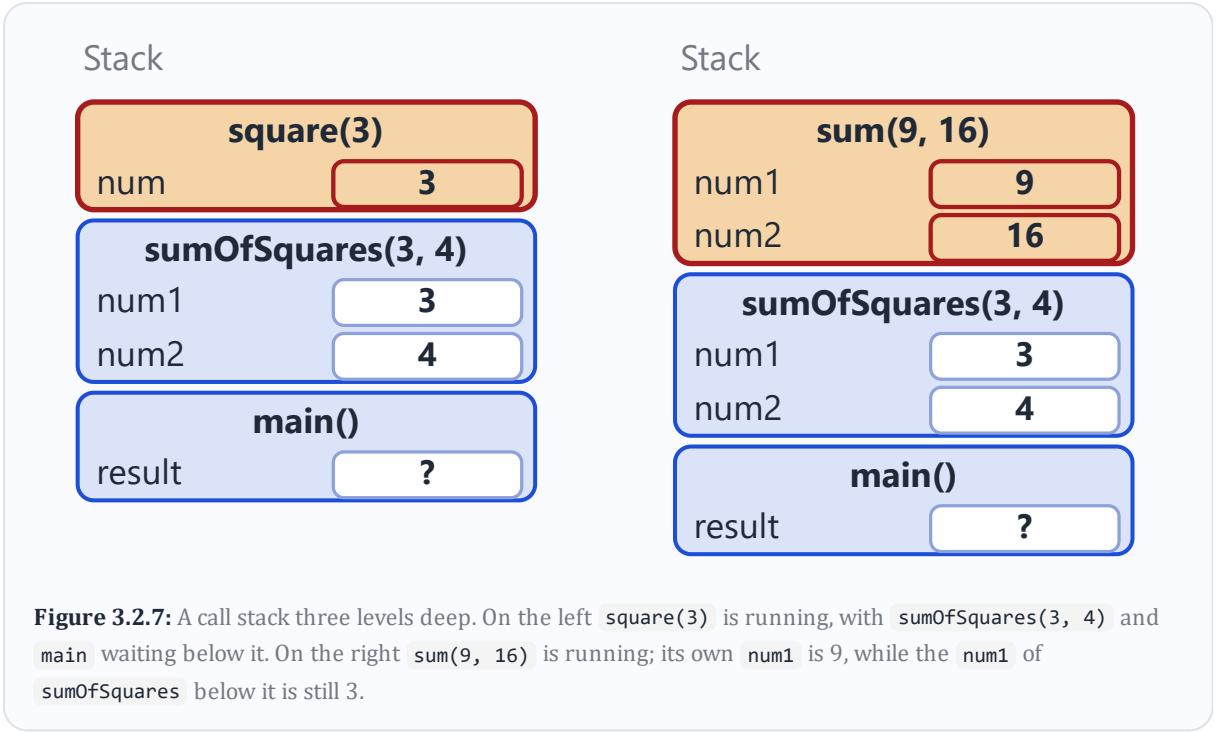


Figure 3.2.7: A call stack three levels deep. On the left `square(3)` is running, with `sumOfSquares(3, 4)` and `main` waiting below it. On the right `sum(9, 16)` is running; its own `num1` is 9, while the `num1` of `sumOfSquares` below it is still 3.

In the fourth step the frame of `square` is closed. The first argument is now known: 9. In the fifth step a new frame is opened for `square(4)` and 16 is returned. This frame is not a reuse of the earlier `square` frame; it is a separate frame opened for a new call of the same method. Every call carries its own parameters and local variables.

In the sixth step the arguments are ready and `sum(9, 16)` is called. The parameters of `sum` are also named `num1` and `num2`. In the frame of `sum`, `num1 = 9`; at the same time, in the `sumOfSquares` frame one level down the stack, `num1 = 3`. The two variables do not affect each other. In the seventh step the 25 returned by `sum` also becomes the return value of `sumOfSquares`. In the eighth step control returns to `main` and `result = 25`. In the ninth step 25 is printed.

During the trace there were at most three frames on the stack at the same time: `main`, `sumOfSquares` and the helper method running at that moment. The depth of the stack equals the longest chain of nested calls. `square` and `sum` were never on the stack at the same time, because neither started before the other had finished.

It is also worth knowing that the stack is not unlimited. Every frame takes up memory. If a method calls itself endlessly, the frames pile up and the stack fills; the program stops with a `StackOverflowError`. Methods that call themselves are called recursive methods; for such methods the depth of the stack must be checked separately.

3.2.8 Built-in methods

Not every programmer writes every job from scratch. Java comes with a standard library of thousands of classes and methods. There are ready-made methods for taking square roots, printing with formatting, sorting arrays and reading files. Using these methods is no different from using the methods we write ourselves: their names, parameters and return types are known, and their insides are not.

3.2.8.1 The Java library and packages

The standard library of Java is called the Java API (Application Programming Interface). The classes in the library are divided into **packages**. A package is a namespace that groups related classes together. Table 3.2.8 lists the most commonly used classes and their packages.

Area	Classes	Example methods	Package
Printing	`System.out`	`print`, `println`, `printf`	`java.lang`
Mathematics	`Math`	`sqrt`, `abs`, `max`, `pow`	`java.lang`
Text	`String`	`length`, `charAt`, `substring`, `equals`	`java.lang`
Arrays	`Arrays`	`sort`, `copyOf`, `binarySearch`, `equals`	`java.util`
User input	`Scanner`	`nextInt`, `nextLine`, `close`	`java.util`
Files	`File`, `FileWriter`	`createNewFile`, `write`, `close`	`java.io`

Area	Classes	Example methods	Package
Database	`DriverManager`, `Statement`	`getConnection`, `executeQuery`, `executeUpdate`	`java.sql`
Data structures	`ArrayList`	`add`, `remove`, `get`, `size`	`java.util`
Time	`LocalDateTime`, `DateTimeFormatter`	`now`, `ofPattern`, `format`	`java.time`
Web / graphics	`URI`, `URL`, `JFrame`	`URI.create`, `openConnection`, `setVisible`	`java.net`, `javax.swing`

Table 3.2.8: The most commonly used classes and methods of the Java API.

The `java.lang` package is special: it is included in every Java program automatically. Since `System`, `Math` and `String` are in this package, they are used without any preparation. For classes in other packages an **import** declaration is written at the top of the file: `import java.util.Scanner;`. If this declaration is left out, the compiler does not recognize the name `Scanner` and says `cannot find symbol`.

Some of the methods in the table are `static` and are called with the class name: `Math.sqrt(16)`, `Arrays.sort(arr)`. Others are called through an object: `in.nextInt()`, `text.length()`. Which form a method is called in is stated in the header of the method in the official Java documentation.

3.2.8.2 Printing to the screen

`System.out` is a ready-made object that represents standard output, that is, the console. Three of its methods are used very often.

```
public class Printing {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
        System.out.print("One ");
        System.out.print("line");
        System.out.println();
        int age = 20;
        System.out.println("Age: " + age);
        System.out.printf("Pi is about %.2f%n", Math.PI);
    }
}
```

```
Hello, World!
One line
Age: 20
Pi is about 3.14
```

- `System.out.println(x)` prints `x` and moves to the next line. `println()` without an argument prints only a line break.
- `System.out.print(x)` prints `x` but stays on the same line. In the program `print("One ")` and `print("line")` run one after the other, and then `println()` ends the line: `One line`.
- `System.out.printf(format, values...)` prints with formatting. The values are put in place of the placeholders in the format string.

Text and a value are joined with the `+` operator: the value of the expression `"Age: " + age` is the string `"Age: 20"`. Because the `+` operator works from left to right, the order of concatenation changes the result. This detail is a point where beginners often get stuck:

```
import java.util.Locale;

public class Format {
    public static void main(String[] args) {
        double avg = 81.66666666666667;
        System.out.printf("%.2f%n", avg);
        System.out.printf(Locale.US, "%.2f%n", avg);
        System.out.printf("%d students, average %.1f, grade %s%n", 3, avg, "BB");
        System.out.printf("[%5d] [%-5d] [%05d]%n", 42, 42, 42);
        System.out.println(1 + 2 + " total");
        System.out.println("total " + 1 + 2);
        System.out.println("total " + (1 + 2));
        System.out.println(String.format("%.3f", Math.PI));
    }
}
```

```
81,67
81.67
3 students, average 81,7, grade BB
[   42] [42   ] [00042]
3 total
total 12
total 3
3,142
```

In the expression `1 + 2 + " total"`, `1 + 2` is first the sum of two numbers: 3. Then 3 is appended to the text: `3 total`. In `"total " + 1 + 2`, however, the first operation is the concatenation of a string with a number: `"total 1"`. Since the result is now a string, the 2 is also appended to the text: `total 12`. To compute the sum as a number, parentheses are needed: `"total " + (1 + 2)`.

The placeholders most often used in `printf` are given in Table 3.2.9.

Placeholder	Meaning	Example	Output
`%d`	integer	`printf("%d", 42)`	`42`

Placeholder	Meaning	Example	Output
`%.2f`	decimal number, 2 digits	`printf("%.2f", Math.PI)`	`3.14`
`%s`	text	`printf("%s", "BB")`	`BB`
`%n`	line break	`printf("a%nb")`	two lines
`%5d` / `%-5d`	right / left aligned in a field of 5 characters	`printf("[%5d]", 42)`	`[ 42]`
`%05d`	pad with zeros instead of spaces	`printf("[%05d]", 42)`	`[00042]`

Table 3.2.9: `printf` placeholders (run with the US English locale).

When `printf` prints a decimal number, it uses the computer's **locale**. On a system set to Turkish the decimal separator is a comma, so `Math.PI` is printed as `3,14`. `println`, on the other hand, ignores the locale and always uses a point: `3.141592653589793`. If a program prints the same number in the two ways, two different forms appear. If the decimal separator should be a point on every system, the locale is given explicitly: the output of `printf(Locale.US, "%.2f%n", avg)` is `81.67`. The `Locale` class is in the `java.util` package and has to be imported.

The placeholder `%.2f` rounds the number rather than truncating it: the value 81.666... becomes `81,67`. `String.format` uses the same format rules as `printf`, but instead of printing the result it returns it as a string. The `3,142` on the last line comes from this method.

3.2.8.3 The Math class

The `Math` class contains ready-made methods for mathematical operations. All of its methods are `static`; so no `Math` object is created, and the methods are called with the class name.

```
double root = Math.sqrt(16);
double absValue = Math.abs(-17.2);
double maximum = Math.max(3, 5);
System.out.println(root);
System.out.println(absValue);
System.out.println(maximum);
System.out.println(Math.pow(2, 10));
System.out.println(Math.round(2.6));
System.out.println(Math.min(-4, 7));
System.out.println(Math.max(3, 5));
```

```
4.0
17.2
5.0
1024.0
3
-4
5
```

The form of each line in the output reveals the return type of the method. Table 3.2.10 lists these methods together with their return types.

Method	Role	Return type	Example → result
`sqrt(x)`	square root	always `double`	`Math.sqrt(16)` → `4.0`
`pow(a, b)`	a to the power b	always `double`	`Math.pow(2, 10)` → `1024.0`
`abs(x)`	absolute value	type of the argument	`Math.abs(-17.2)` → `17.2`
`max(a, b)`, `min(a, b)`	the larger / the smaller	common type of the arguments	`Math.max(3, 5)` → `5`
`round(x)`	rounds to the nearest integer	`long` for `double`	`Math.round(2.6)` → `3`
`floor(x)`, `ceil(x)`	rounds down / up	`double`	`Math.floor(-2.5)` → `-3.0`
`PI`	the constant π (not a method)	`double`	`Math.PI` → `3.141592653589793`

Table 3.2.10: Commonly used `Math` methods and their return types.

`abs`, `max` and `min` are overloaded; different versions run depending on the type of the argument. `Math.max(3, 5)` takes two `int`s and returns an `int`: `5`. When the result of the same call is assigned to a `double` variable in a program, it is widened to 5.0 and `5.0` is printed. `sqrt` and `pow`, on the other hand, have only a `double` version; even if an integer argument is given, the result is a `double`. Assigning the result directly to an `int` variable is therefore a compile error:

```
public class SqrtInt {
    public static void main(String[] args) {
        int k = Math.sqrt(16);
    }
}
```

```
SqrtInt.java:3: error: incompatible types: possible lossy conversion from double to int
        int k = Math.sqrt(16);
                         ^
1 error
```

If the result is needed as an integer, an explicit cast is written: `int k = (int) Math.sqrt(16);`. The `(int)` cast truncates the fractional part; it does not round. In expressions where return types are mixed, the result must be followed carefully. The following program gives a few more examples.

```
public class MathExtras {
    public static void main(String[] args) {
        System.out.println(Math.round(2.5) + " " + Math.round(-2.5) + " " +
Math.round(-2.6));
        System.out.println(Math.floor(-2.5) + " " + Math.ceil(-2.5));
        System.out.println(Math.sqrt(-4));
        System.out.println((int) Math.pow(2, 10));
        System.out.println(Math.max(Math.abs(-8), Math.sqrt(49)));
        System.out.println(Math.sqrt(16) + Math.max(3, 7) / 2);
        System.out.println(Math.PI);
        System.out.println(Math.hypot(3, 4));
        long r = Math.round(2.6);
        System.out.println(r);
    }
}
```

```
3 -2 -3
-3.0 -2.0
NaN
1024
8.0
7.0
3.141592653589793
5.0
3
```

The first line shows the behavior of `Math.round`. The method adds 0.5 to the number and rounds down. For 2.5 the result is 3; for −2.5, −2.5 + 0.5 = −2. So halves are always rounded toward positive infinity; −2.6, however, becomes −3. On the second line `floor` rounds −2.5 down to −3.0, and `ceil` rounds it up to −2.0. The square root of a negative number is undefined over the real numbers. `Math.sqrt(-4)` does not raise an error; it returns the special value `NaN` (Not a Number).

In the expression `Math.max(Math.abs(-8), Math.sqrt(49))` on the fifth line, the inner calls run first. `Math.abs(-8)` gives an `int`: 8. `Math.sqrt(49)` gives a `double`: 7.0. Because the arguments of `max` are `(int, double)`, the overloading rules come into play and the version `max(double, double)` is chosen. The result is `8.0`, not `8`.

In the expression `Math.sqrt(16) + Math.max(3, 7) / 2` on the sixth line, operator precedence matters. The division is done before the addition. `Math.max(3, 7)` returns an `int`: 7. `7 / 2` is an integer division: 3. Then `4.0 + 3` is computed and the result is `7.0`. `Math.hypot(3, 4)` gives the hypotenuse of the triangle whose legs are 3 and 4: 5.0.

3.2.9 Scope

A variable cannot be used everywhere in a program. Where it is defined determines where it can be seen and how long it lives. These rules let methods work independently of each other: variable names inside one method do not get mixed up with names in other methods.

3.2.9.1 Local variables

Definition 3.2.9

A **local variable** is a variable defined inside a method or a block that is valid only there. The parameters of a method are also local variables. A local variable exists only while the method is running; when the method returns, it is removed.

In the following code the `multiply` method uses a local variable named `result`. After calling `multiply`, `otherMethod` tries to reach this variable.

```
public int multiply(int num1, int num2) {
    int result = num1 * num2; // 'result' is a local variable
    return result;
}
public void otherMethod() {
    multiply(2, 3);
    System.out.println(result);  // ERROR
}
```

```
LocalOutside.java:8: error: cannot find symbol
        System.out.println(result);  // ERROR
                           ^
  symbol:   variable result
  location: class LocalOutside
1 error
```

The compiler cannot find the name `result` inside `otherMethod`. `result` belongs to `multiply` and exists only while `multiply` is running. Figure 3.2.8 shows this in memory: while `multiply(2, 3)` runs, its frame holds `num1`, `num2` and `result`. When the method returns, the frame and the variables in it are removed.

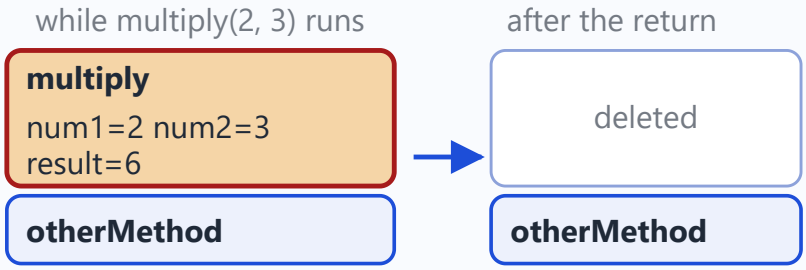


Figure 3.2.8: The life of a local variable. On the left, while `multiply(2, 3)` runs, its frame is on top of the stack: `num1 = 2`, `num2 = 3`, `result = 6`. On the right the method has returned; the frame is gone and only `otherMethod` is left below.

The way to carry the value out of the method is again `return`. If `int s = multiply(2, 3);` is written inside `otherMethod`, the returned 6 is assigned to `s`, a local variable of `otherMethod` itself. In the code that causes the compile error, on the other hand, the line `multiply(2, 3);` assigns the result nowhere, so the 6 is lost.

Local variables have one more rule: they must be given a value before they are used. Class fields and array elements have default values (0, `false`, `null`); local variables do not. Reading a local variable that has not been given a value is a compile error:

```
public class UninitializedLocal {
    public static void main(String[] args) {
        int t;
        System.out.print(t);
    }
}
```

```
UninitializedLocal.java:4: error: variable t might grade have been initialized
        System.out.print(t);
                         ^
1 error
```

3.2.9.2 Scope rules and nested blocks

Definition 3.2.10

The **scope** of a variable is the region of the program in which its name is valid. In Java the scope of a local variable begins at the line where it is declared and ends at the end of the braced block in which it is declared.

Scope determines two things: where the variable can be **accessed** and how long it will **live**. In Java, blocks can be nested. The body of the class is the outermost block. The methods are inside it. Inside the methods there can in turn be `if` blocks, loops, or blocks written as plain braces. Figure 3.2.9 shows these layers.

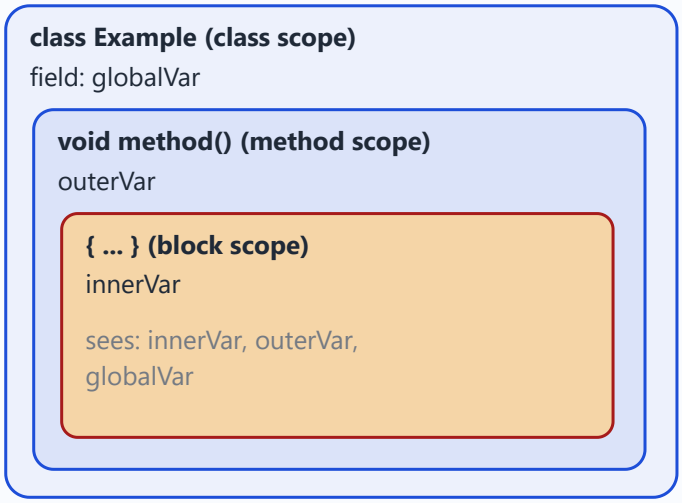


Figure 3.2.9: Nested scopes. On the outside is the class scope with its field `globalVar`; inside it the method scope with `outerVar`; innermost the block scope with `innerVar`. The innermost block sees all the variables outside it.

The rule is one-way: an inner scope sees the variables of the outer scope; an outer scope cannot see the variables inside. In the following program the inner block reaches the variables of both the method and the class.

```
public class InnerScope {
    static int globalVar = 10; // global (class field)
    static void method() {
        int outerVar = 10;       // local scope
        {
            int innerVar = 5;
            // outerVar and innerVar are accessible here
            System.out.println(outerVar + innerVar
                    + globalVar);
        }
        // innerVar does NOT exist here
    }
    public static void main(String[] args) {
        method();
    }
}
```

```
25
```

The `println` in the inner block sees all three variables: `innerVar` is in its own block, `outerVar` in an outer block, and `globalVar` in the class scope. The total is 10 + 5 + 10 = 25. After the closing brace of the block, `innerVar` no longer exists. If `innerVar` is used there, the compiler says `cannot find symbol`.

`globalVar`, defined in the class body outside the methods, is a **field**. All methods of the class see this field. What programming languages call a "global variable", one that is accessible everywhere and lives for the whole program, corresponds in Java to `static` fields. In Java a variable cannot be defined outside a class; there is no truly global variable.

Scope	Where is it defined?	Where can it be accessed?	How long does it live?
Local (block)	Inside a block	From the line where it is defined to the end of the block	As long as the block runs
Local (method)	In the method body or the parameter list	From inside the method	As long as the method runs
`static` field	In the class body, outside the methods	From all methods of the class	For the whole program

Table 3.2.11: Kinds of scope.

Loop variables obey this rule too. The scope of `i`, defined in the header `for (int i = 0; ...)`, is the loop itself. Using `i` after the loop has ended is a compile error:

```
public class OutsideLoop {
    public static void main(String[] args) {
        for (int i = 0; i < 3; i++) { }
        System.out.print(i);
    }
}
```

```
OutsideLoop.java:4: error: cannot find symbol
        System.out.print(i);
                         ^
  symbol:   variable i
  location: class OutsideLoop
1 error
```

A benefit of limited scope is that the same name can be reused in blocks that cannot see each other. Two sibling blocks, that is, blocks where one ends before the other begins, can define variables with the same name without confusion. That is why it is valid to use `i` in two `for` loops written one after

the other:

```
public class SiblingBlocks {
    public static void main(String[] args) {
        { int t = 1; System.out.print(t + " "); }
        { int t = 2; System.out.println(t); }
        for (int i = 0; i < 2; i++) { int square = i * i; System.out.print(square + " "); }
        for (int i = 5; i < 7; i++) { System.out.print(i + " "); }
        System.out.println();
    }
}
```

```
1 2
0 1 5 6
```

3.2.9.3 Name clashes and shadowing

Using the same name in different scopes has two different consequences. Java forbids one of them and allows the other.

The forbidden case: an inner block cannot use the same name as a *local* variable in an outer block. Since both variables would live at the same time and carry the same name, it would be unclear which one is meant. Java does not allow this ambiguity:

```
public void method() {
    int outerVar = 10;
    {
        int outerVar = 5;  // ERROR
    }
}
```

```
SameName.java:5: error: variable outerVar is already defined in method method()
            int outerVar = 5;  // ERROR
                ^
1 error
```

The same rule applies to parameters, because parameters are also local variables. A variable with the same name as a parameter cannot be defined in the body of the method:

```
public class SameNameParam {
    static int f(int x) {
        int x = 3;
        return x;
    }
}
```

```
SameNameParam.java:3: error: variable x is already defined in method f(int)
        int x = 3;
            ^
1 error
```

The allowed case: a local variable may have the same name as a *field* of the class. In this case the local variable **shadows** the field.

Definition 3.2.11

Shadowing is when a name in an inner scope hides the same name in an outer scope. When a local variable shadows a field, the name refers to the local variable within that scope. The shadowed `static` field can be reached with the class name: `ClassName.area`.

```
public class Shadow {
    static int value = 10;  // class field
    static void show() {
        int value = 5;      // shadows the field
        System.out.println(value);
        System.out.println(Shadow.value);
    }
    public static void main(String[] args) {
        show();
        System.out.println(value);
    }
}
```

```
5
10
10
```

When Java sees a name, it first looks in the innermost scope and, if it does not find it, moves out to the next scope. Inside `show`, the name `value` is found in the local variable and 5 is printed. The notation `Shadow.value`, on the other hand, directs the search straight to the field of the class, and 10 is

printed. In `main` there is no local variable named `value`; the search moves out to the class scope and the field is found: 10. Table 3.2.12 traces these steps.

Step	Line	Call stack	Variables	Output
1	9: `show();`	main()	Shadow.value=10	—
2	3: `static void show() {`	main() → show()	Shadow.value=10	—
3	4: `int value = 5; // shadows the field`	main() → show()	Shadow.value=10, value (local)=5	—
4	5: `System.out.println(value);`	main() → show()	Shadow.value=10, value (local)=5	`5`
5	6: `System.out.println(Shadow.value);`	main() → show()	Shadow.value=10, value (local)=5	`10`
6	10: `System.out.println(value);`	main()	Shadow.value=10	`10`

Table 3.2.12: Tracing shadowing. `Shadow.value` denotes the class field, and "value (local)" denotes the local variable of `show`.

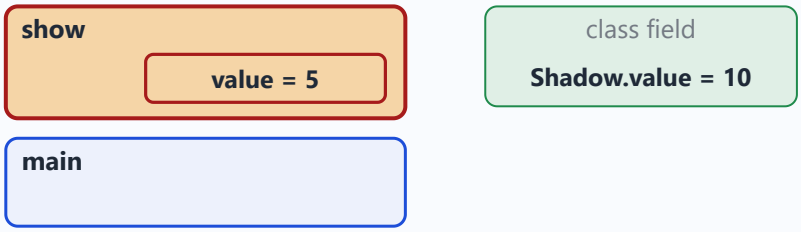


Figure 3.2.10: The moment of shadowing. The class field `Shadow.value = 10` lives outside the stack, for the whole program. In the frame of `show` there is a local `value = 5` with the same name; inside the method the name `value` refers to this local variable.

Shadowing does not cause a compile error, but it can lead to insidious bugs. A programmer may think they are changing the field while actually changing a local copy. There is also a common place where shadowing is used deliberately: when working with objects, in constructors and `set` methods the parameter has the same name as the field, and the field is reached with `this.area`.

3.2.9.4 Lifetime

Scope says *where* a variable is visible; lifetime says *when* it exists. Local variables are born when their block is entered and die when the block is left. `static` fields are created once when the program starts and live until the program ends. The following program uses both kinds of variable together.

```
static int callCount = 0;  // class field
static int multiply(int num1, int num2) {
  callCount++;
  int result = num1 * num2;
  return result;
}
public static void main(String[] args) {
  int first = multiply(2, 3);
  {
    int second = multiply(first, 4);
    System.out.println(second);
  }
  System.out.println(first + " " + callCount);
}
```

```
24
6 2
```

Step	Line	Call stack	Variables	`callCount`	Output
1	8: `int first = multiply(2, 3);`	main()	first=?	0	—
2	3: `callCount++;`	main() → multiply(2, 3)	num1=2, num2=3	1	—
3	4: `int result = num1 * num2;`	main() → multiply(2, 3)	num1=2, num2=3, result=6	1	—
4	8: `int first = multiply(2, 3);`	main()	first=6	1	—
5	4: `int result = num1 * num2;`	main() → multiply(6, 4)	num1=6, num2=4, result=24	2	—
6	10: `int second = multiply(first, 4);`	main()	first=6, second=24	2	—
7	11: `System.out.println(second);`	main()	first=6, second=24	2	`24`
8	13: `System.out.println(first + " " + callCount);`	main()	first=6	2	`6 2`

Table 3.2.13: The birth and death of variables. The `callCount` column shows the current value of the class field.

When the program starts, the field `callCount` is created and gets the value 0. In the first step `multiply(2, 3)` is called. In the second step the method increases the class field by 1. Methods can reach the class field because the field is in class scope. In the third step the local `result` is born and becomes 6. In the fourth step the method returns; `num1`, `num2` and `result` disappear. The returned 6 is assigned to `first`, a local variable of `main`.

In the fifth step `multiply(first, 4)` is called from inside the inner block. In the new frame `result = 24` and the field rises to 2. This `result` is not the `result` of the first call; the `result` of the first call was removed long ago. In the sixth step `second` is born. Because `second` is defined in the inner

block, it lives only in that block. Figure 3.2.11 shows this moment.

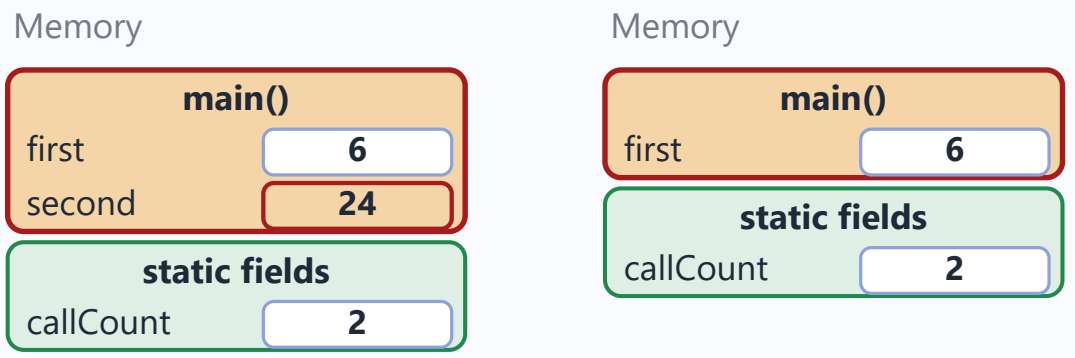


Figure 3.2.11: Lifetime. On the left the inner block is running: in the frame of `main` there are `first = 6` and `second = 24`; the class field `callCount = 2` is in a separate region. On the right the block has ended: `second` has disappeared, while `first` and the field live on.

In the seventh step `second` is printed inside the block: 24. In the eighth step the block has ended and `second` no longer exists. `first`, however, lives until `main` ends. The last line prints `6 2`: `first` is 6 and `multiply` has been called twice.

It is worth noticing that in this example a class field is used as a "counter". A local variable could not do this job: a counter defined inside `multiply` is born anew on every call and starts from 0. Information that has to be kept between calls is held not in a local variable but in a field. Even so, fields should be used with moderation. A field that every method can change makes the behavior of the program hard to follow.

3.2.10 Abstraction and design

Getting a method to compile and run correctly is half the job. The other half is designing the method well: how much work it does, which inputs it takes, and what it is called. Well-designed methods get reused in other programs, are easy to test, and can still be understood years later. This section covers the ideas that design rests on, namely abstraction, generalization and decomposition, followed by practical principles.

3.2.10.1 Abstraction and the black box

Definition 3.2.12

Abstraction means hiding the details and showing only the important features in order to reduce complexity.

Abstraction is everywhere in daily life. To drive a car you do not need to know what is inside the engine. The steering wheel, the pedals and the gear lever are an interface through which the driver talks to the car. Even if the engine changes, for example when an electric motor replaces a petrol engine, the driver uses the same steering wheel and the same pedals.

Methods are a kind of abstraction as well. In Figure 3.2.12 `Math.sqrt` is drawn as a black box. The user gives the box 16 and gets 4.0. How the square root is computed inside the box is hidden and does not concern the user.

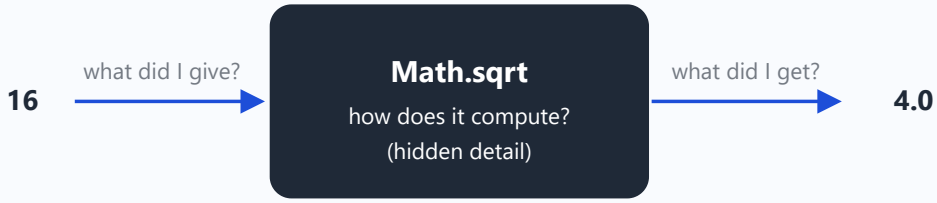


Figure 3.2.12: A black box. The user knows only the input (“what did I give?” 16) and the output (“what did I get?” 4.0). How `Math.sqrt` computes the result is a hidden detail.

Abstraction has two benefits. First, it makes the program understandable. A large program is split into pieces that can each be thought about separately; whoever uses a piece does not have to know its inside. Second, it makes change easier. When the inside of a method changes, the code that uses it is not affected as long as what the method does stays the same.

This kind of abstraction over computations is called **functional abstraction** or **procedural abstraction**. The basic idea is that what a function computes matters more than *how* it computes it. The user knows only the signature and the purpose of the method:

```
// The user knows only the signature:
static boolean isPrime(int num)
// "returns true if num is prime"
if (isPrime(17)) { ... }
```

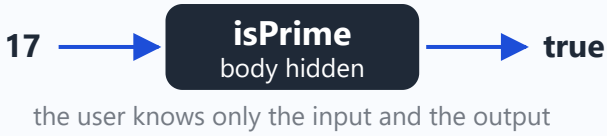


Figure 3.2.13: `isPrime` as a black box. 17 goes in and `true` comes out. The body is hidden; the user knows only the input and the output.

All a programmer who writes the call `isPrime(17)` needs to know is that the method returns, as a `boolean`, whether a number is prime. Does the method divide the number by every number starting from 2, or does it try only up to the square root? This question is irrelevant to the user of the method. If a faster algorithm is written tomorrow, only the body of `isPrime` changes; the line `if (isPrime(17))` stays exactly as it is.

The simplest example is again `System.out.println("Hello")`. This method writes the given text to the console. Details such as which encoding the characters are sent in, or which operating system call is used, are hidden from the programmer. Without abstraction, one would need to know all these details just to print a single line to the screen.

For abstraction to work, the purpose of the method must be clear. Since the user will not look at the body, the name, parameters and return type of the method must tell fully what it does. The header `isPrime(int num)` does this. The header `process2(int a)`, on the other hand, tells nothing; the user has to read the body, and the abstraction collapses.

3.2.10.2 Generalization

The meaning of a piece of code depends on its surroundings. The value of a number is related to the other terms in an equation, and the meaning of a word to its place in a sentence. In code, too, what a statement does depends on which variables hold which values in the block it is in. These surroundings are called the **context**.

Definition 3.2.13

Generalization means making a solution that belongs to one context usable in other contexts too. In code the tool of generalization is the parameter: constant values embedded in the solution are turned into parameters.

```java
public class Generalization {
    // Specific: prints exactly 10 '-' characters
    static void drawLine() {
        System.out.println("----------");
    }
    // General: length and character are parameters
    static void drawLine(int length, char ch) {
        for (int i = 0; i < length; i++) {
            System.out.print(ch);
        }
        System.out.println();
    }
    public static void main(String[] args) {
        drawLine();
        drawLine(5, '*');
        drawLine(12, '=');
    }
}
```

```
----------
*****
============
```

The parameterless `drawLine` does only one thing: it prints ten dashes. The length and the character are embedded in the method. When a line of five stars is needed, this method is of no use; a new method has to be written. The version with two parameters generalizes the same job. The embedded number 10 has become the parameter `length`, and the embedded character `'-'` has become the parameter `ch`. Now a single method draws a line of any length with any character: `drawLine(5, '*')` prints five stars and `drawLine(12, '=')` prints twelve equals signs.

The two methods in the example have the same name; this is an example of overloading. The specific version could also have been written in terms of the general one: its body would be the one-line call `drawLine(10, '-');`. That way the logic of drawing a line stays in one place.

Generalization has a limit. Turning every constant into a parameter makes the method unusable. Calling a method with ten parameters can be harder than doing the job by hand. Generalization is limited to the values that are really expected to change.

3.2.10.3 Decomposition

Definition 3.2.14

Decomposition means splitting a large problem into small, independent subproblems. In programming each subproblem becomes a method.

Consider a program that reads three grades of a student and prints the average and the letter grade. The problem splits into three subtasks: reading the grades, computing the average, and finding the letter grade from the average. Figure 3.2.14 shows this structure.

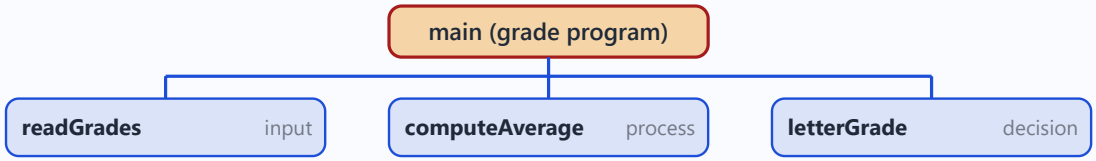


Figure 3.2.14: The parts of the grade program. `main` calls three submethods: `readGrades` (input), `computeAverage` (process) and `letterGrade` (decision).

In the following program all four methods are written out in full. The thresholds in `letterGrade` are chosen according to a common letter-grade table; a different table requires changing only this method.

```
import java.util.Scanner;

public class GradeProgram {
    // readGrades and letterGrade are separate methods too
    static int[] readGrades(Scanner in, int n) {
        int[] grades = new int[n];
        for (int i = 0; i < n; i++) {
            grades[i] = in.nextInt();
        }
        return grades;
    }
    static double computeAverage(int[] grades) {
        int sum = 0;
        for (int n : grades) {
            sum += n;
        }
        return (double) sum / grades.length;
    }
    static String letterGrade(double avg) {
        if (avg >= 90) return "AA";
        if (avg >= 85) return "BA";
        if (avg >= 80) return "BB";
        if (avg >= 75) return "CB";
        if (avg >= 70) return "CC";
        if (avg >= 60) return "DC";
        if (avg >= 50) return "DD";
        return "FF";
    }
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int[] grades = readGrades(in, 3);
        double avg = computeAverage(grades);
        System.out.println("Average: " + avg + " -> " + letterGrade(avg));
    }
}
```

The program was run with two different inputs:

```
Average: 81.66666666666667 -> BB
```

```
Average: 52.333333333333336 -> DD
```

Each method does a single job. `readGrades` takes the `Scanner` object and the number of grades to read as parameters, and returns the grades it read as an array. The `Scanner` object is created once in `main` and passed to the method; this way the reading method does not have to know where the input comes from. `computeAverage` takes the array and returns the average. The explicit cast in the

expression `(double) sum / grades.length` prevents integer division. 80 + 90 + 75 = 245; 245 / 3 would be 81 with integer division and 81.666… with floating-point division. `letterGrade` looks at the average and returns a `String`. The early-`return` pattern here takes the place of an `else if` chain: the conditions are tried from top to bottom, and the method ends at the first one that holds.

`main` now reads like a table of contents: read, compute, print. What is done is visible in `main`; how it is done is in the submethods. When a bug appears, it is also clear which method to look at. If the average is wrong, look at `computeAverage`; if the letter grade is wrong, look at `letterGrade`.

Another benefit of decomposition is that the methods can be tested separately. `letterGrade` can be tested without touching the keyboard at all: `letterGrade(85)` should return “BA” and `letterGrade(84.99)` should return “BB”. If reading and computing had been written intertwined, every test would require typing grades at the keyboard.

3.2.10.4 When to write a method: the DRY principle

The clearest sign that a piece of code should be a method is repetition. The name of this principle is **DRY** (Don't Repeat Yourself). The moment you feel the need to copy and paste code, that code is a candidate to become a method.

The first fragment below is the repetitive version and the second is the version with a method. Both print the average of two students.

```
double avg1 = (70 + 80 + 90) / 3.0;
System.out.println(avg1);
double avg2 = (55 + 60 + 100) / 3.0;
System.out.println(avg2);
// if the formula changes, it has to be
// fixed separately in every place
```

```
static double average(int a, int b, int c) {
    return (a + b + c) / 3.0;
}
System.out.println(average(70, 80, 90));
System.out.println(average(55, 60, 100));
```

```
80.0
71.66666666666667
80.0
71.66666666666667
```

In the repetitive version the average formula is written twice. If the formula changes, for example if the grades are to be weighted differently, both places have to be fixed separately. In real programs the same formula can appear in dozens of places, and forgetting one is easy. In the version with a method the formula is in one place: the body of the `average` method. The change is made once and shows up in every call.

The value `71.66666666666667` in the output comes from the limited precision of `double`. 215 / 3 = 71.666... has an infinite decimal expansion. `double` stores it with about 16 significant digits and the last digit is rounded. In a result to be shown to a user, the number of digits is limited with `printf("%.2f", ...)`.

Repetition is not the only criterion. A piece of code that is used only once can also deserve to be a method. There are two reasons: the piece does a meaningful job on its own, or gathering it under a name makes the calling code readable. `letterGrade` in the grade program is called only once, but being a separate method simplifies `main` and makes testing easier.

3.2.10.5 Designing good methods

The properties of a good method can be gathered into six principles.

Single responsibility
: Each method does only one job. If the task of a method is described with the word “and” (read *and* compute *and* print), the method should be split.

Descriptive name
: The name of a method is a verb that tells what it does: `computeAverage`, `isPrime`, `readGrades`. Names such as `process2` say nothing.

Parameters
: A method takes the data it needs through parameters; it does not secretly depend on outside variables. A small number of parameters improves readability.

Return value
: The result is returned explicitly with `return`. A method that computes does not print; printing is done by the caller.

Short and focused
: Methods of a few lines that fit on one screen are easy to understand and easy to test.

Called where needed
: A method is called only where it is needed. Computing the same result again and again in a loop is wasted work; the result is computed once and kept in a variable.

The principle “a method that computes does not print” is the one most often broken. The design that first comes to a student's mind is usually this:

```
static void computeAndPrintAverage(int a, int b, int c) {
    System.out.println((a + b + c) / 3.0);
}
```

This method can only print the average to the screen. Code that wants to test whether the average is greater than 50, save the average to a file, or include it in the class average cannot use it; the result has stayed on the screen. A better design is to return the average and let the caller decide what to do:

```
static double average(int a, int b, int c) {
    return (a + b + c) / 3.0;
}
// the caller decides:
System.out.println(average(70, 80, 90));
if (average(40, 50, 45) < 50) System.out.println("failed");
```

The principle “do not secretly depend on outside variables” rests on the same reasoning. A method that reads its values from class fields depends on the current values of the fields. To understand what the method will return, you have to know where in the rest of the program the fields are changed. The result of a method that takes parameters, on the other hand, depends only on its arguments; with the same arguments it gives the same result every time.

3.2.10.6 Naming

Choosing good names applies not only to methods but also to variables. Each variable should represent one thing. Using the same variable first for an age and then for a total confuses the reader and leads to logic errors. The name of a variable should tell what kind of data it holds.

```
int studentCount;      // a good name
int a;                  // a meaningless name
int age = 25;
String name = "Ahmet";
double average = 85.5;
String bookTitle;
```

Meaningless or single-letter names reduce readability. There are two exceptions: conventional names such as `i` and `j` as loop counters, and `x` and `y` in mathematical formulas. Table 3.2.14 compares a few bad names with better ones.

Bad	Good	Why?
`int x1, x2;`	`int midtermGrade, finalGrade;`	The name tells which grade is which.
`double d;`	`double circleArea;`	It is clear what the variable holds the value of.
`boolean f;`	`boolean isPrime;`	`if (isPrime)` reads like a question.

Table 3.2.14: Bad and good variable names.

Java's writing conventions also reveal the kind of a name. Variable and method names are written in camelCase and start with a lowercase letter: `studentCount`, `readGrades`. Class names start with an uppercase letter: `Book`, `Scanner`. Constants that never change are written in uppercase letters with underscores: `MAX_VALUE`. These conventions are not required by the compiler, but they are universal among Java programmers. A reader who sees a name immediately understands whether it is a variable, a class or a constant.

Using non-ASCII characters, such as Turkish letters, in variable and method names is technically allowed in Java: `int öğrenciSayısı;` compiles. But problems arise when the encoding of the file and the encoding the compiler expects do not match. For this reason, in courses and in most projects names are written with the letters of the English alphabet: `studentCount`.

3.2.11 Worked problems

The following problems use method calls, pass by value and overloading together. All solutions were run with Java.

Example 3.2.1

What does the following program print? Why does `a` not change? How many frames are on the stack at most while `cube(2)` is running?

```
static int f(int x) {
    x = x * 2;
    return x + 1;
}
public static void main(String[] args) {
    int a = 3;
    int b = f(a);
    System.out.println(a + " " + b);
    System.out.println(f(f(1)));
}
```

```
static int square(int n) { return n * n; }
static int cube(int n) { return n * square(n); }
// inside main:
System.out.println(cube(2) + square(3));
```

```
3 7
7
17
```

In the call `f(a)`, the value of `a`, that is 3, is copied into the parameter `x` of `f`. `x = x * 2` makes the copy 6, and `x + 1`, that is 7, is returned. `b` becomes 7. `a` does not change, because `f` changed only its copy: the first line is `3 7`.

In the expression `f(f(1))` the inner call runs first: `f(1)` = 1 × 2 + 1 = 3. The outer call runs with this value: `f(3)` = 3 × 2 + 1 = 7. The second line is `7`.

In the expression `cube(2) + square(3)`, `cube(2)` computes `2 * square(2)`. `square(2)` returns 4 and `cube(2)` becomes 8. Then `square(3)` = 9 is computed. The total is 8 + 9 = 17. While `cube(2)` runs, the deepest moment is when `square(2)` is running: there are 3 frames on the stack, `main`, `cube(2)` and `square(2)`. When `square(3)` is called, the frame of `cube` has long been closed; at that moment there are only 2 frames on the stack.

Example 3.2.2

Which `sum` method do the first two `println` calls invoke? Why does the last line print `8.0` and not `8`?

```
static int sum(int a, int b) { return a + b; }
static double sum(double a, double b) { return a + b; }
// inside main:
System.out.println(sum(2, 3.0));
System.out.println(sum(2, 3));
System.out.println(Math.max(Math.abs(-8), Math.sqrt(49)));
```

```
5.0
5
8.0
```

The arguments of the call `sum(2, 3.0)` are `(int, double)`. The `(int, int)` version does not fit, because 3.0 cannot be narrowed to `int`. The `(double, double)` version fits: the value 2 is widened to 2.0. The result is `5.0`. In the call `sum(2, 3)` both versions are applicable; the most specific, `(int, int)`, is chosen and `5` is printed. On the last line, because `Math.sqrt` returns `double`, the `(double, double)` version of `Math.max` is chosen. The larger of 8 and 7.0 is 8.0.

Example 3.2.3

Let us write the following methods: `isEven`, which returns whether a number is even; `largest`, which finds the largest of three numbers with `Math.max`; `power`, which raises to a power without `Math.pow`; `digitSum`, which finds the sum of the digits; `circleArea`, which returns the area of a circle; and `area`, overloaded for a circle and a rectangle.

```java
public class Exercises {
    static boolean isEven(int num) {
        return num % 2 == 0;
    }
    static int largest(int a, int b, int c) {
        return Math.max(a, Math.max(b, c));
    }
    static int power(int base, int exp) {
        int result = 1;
        for (int i = 0; i < exp; i++) result *= base;
        return result;
    }
    static int digitSum(int n) {
        int t = 0;
        while (n != 0) { t += n % 10; n /= 10; }
        return t;
    }
    static double circleArea(double r) {
        return Math.PI * r * r;
    }
    static double area(double r) {
        return Math.PI * r * r;
    }
    static double area(double width, double height) {
        return width * height;
    }
    public static void main(String[] args) {
        System.out.println(isEven(7) + " " + isEven(10) + " " + isEven(-4));
        System.out.println(largest(4, 9, 2));
        System.out.println(power(2, 10) + " " + power(5, 0));
        System.out.println(digitSum(2024) + " " + digitSum(-2024));
        System.out.printf("%.2f%n", circleArea(2));
        System.out.println(area(1) + " " + area(3, 4));
    }
}
```

```
false true true
9
1024 1
8 -8
12,57
3.141592653589793 12.0
```

The body of `isEven` is a single line: `return num % 2 == 0;`. The result of the comparison is already a `boolean`; there is no need to write `if (num % 2 == 0) return true; else return false;`. It also works for negative numbers: `-4 % 2` is 0.

`largest` calls `Math.max` in a nested way: first the larger of `b` and `c` is found, and then this value is compared with `a`. `largest(4, 9, 2)` returns 9. `power` uses an accumulator. `result` starts at 1 and is multiplied by `base` `exp` times. `power(2, 10)` returns 1024. In `power(5, 0)` the loop never runs and 1 is returned; this agrees with the rule $5^0 = 1$ in mathematics.

`digitSum` takes the last digit of the number with `n % 10` and discards it with `n /= 10`. Table 3.2.15 traces the loop for 2024.

Iteration	`n % 10`	`t` (after)	`n / 10`
1	4	4	202
2	2	6	20
3	0	6	2
4	2	8	0
`n = 0`: the loop ends, `return t;` → 8			

Table 3.2.15: Tracing the loop of `digitSum(2024)` (`t` starts at 0).

For negative numbers, since Java's remainder operation takes the sign of the dividend, `digitSum(-2024)` returns −8. If the digit sum should always be positive, `n = Math.abs(n);` is written at the start of the method.

`circleArea(2)` returns π × 2 × 2 ≈ 12.566. `printf("%.2f%n", ...)` rounds this to two digits; with the Turkish locale the output is `12,57`. In the overloaded `area` methods, the number of arguments determines which version is called. `area(1)` runs the circle version and gives π: `3.141592653589793`. `area(3, 4)` runs the rectangle version: `12.0`. Because the integer arguments are widened to `double` parameters, the result is a `double` as well.

3.2.12 Common mistakes

The compiler catches some mistakes with methods and points out their location in its message. Others produce programs that compile but give unexpected results. The second group is more dangerous, because the program runs wrongly without any warning.

Common mistake

Calling a non-`static` method from `main`. Message: `non-static method add(int,int) cannot be referenced from a static context`. Either `static` is added to the header of the method, or an object is created and the method is called through it.

Common mistake

Forgetting `return` on some paths. Message: `missing return statement`. The compiler does not reason about the values of conditions; a chain `if (n > 0) ... if (n <= 0) ...` gives an error even though it looks complete to a human. The last branch becomes a plain `else`, or a default `return` is put at the end of the method.

Common mistake

Using the result of a `void` method. The line `int x = sayHello();` gives `incompatible types: void cannot be converted to int`, and `println(sayHello())` gives `'void' type not allowed here`. A `void` method produces no value; its call can only be a statement on its own.

Common mistake

Missing or extra arguments. Message: `method add in class ... cannot be applied to given types`; below it the lines `required: int,int` and `found: int` show the expected and the supplied types. The number, order and types of the arguments must match the signature.

Common mistake

Using a local variable outside its method. Message: `cannot find symbol ... variable result`. A local variable disappears when the method ends; its value is returned with `return` and the caller assigns it to its own variable.

Common mistake

The same name as an outer local variable in an inner block. Message: `variable outerVar is already defined in method method()`. The same rule applies to parameters. A local variable having the same name as a *field*, on the other hand, is allowed; then shadowing occurs.

Common mistake

Expecting a swap method to work. `swap(int x, int y)` swaps only copies; after the call `swap(arr[0], arr[2])` the array stays the same. To swap array cells, the method takes the array and the indices: `swap(int[] d, int i, int j)`.

Common mistake

Expecting an argument to change. After the call `increase(num)`, `num` stays the same; what changed is the copy in the method. The new value is returned with `return` and assigned: `num = increase(num);`. The compiler gives no warning when the returned value is not assigned.

Common mistake

Overloading by return type. `int f(int a)` and `double f(int a)` cannot coexist: `method f(int) is already defined`. Only the name and the parameter types determine the signature; the return type and the parameter names are not part of it.

Common mistake

Forgetting the type of a `Math` result. The line `int k = Math.sqrt(16);` gives the message `possible lossy conversion from double to int`. `sqrt` and `pow` always return `double`; if an integer is needed, the `(int)` cast is written.

Common mistake

Expecting the return type to change the calculation. The method `static double half(int n) { return n / 2; }` returns 3.0 for 7. The division is done between two `int`s and the fraction is discarded; the `double` return type merely widens the result. The correct form is `n / 2.0`.

Common mistake

Printing inside a method that computes. The result of a method such as `computeAndPrintAverage` stays on the screen; it cannot be used in another calculation. The method returns the result, and the caller does the printing.

3.2.13 Summary

- A **function** is a named piece of code that does one job and can be reused. In Java it is always written inside a class and is called a **method**. The input comes in through parameters, and the output comes back with `return`.

- Method header: `access [static] returnType name(parameters)`. Each parameter carries its own type. The **signature** is the name plus the sequence of parameter types; the return type is not part of it. Methods cannot be defined inside one another.
- A **parameter** is the variable in the definition, and an **argument** is the value in the call. Arguments are evaluated first (left to right) and then copied into the parameters. The number, order and types must match.
- Every call opens a new **frame** on the **call stack**; the frame holds the parameters and local variables. When the method returns, the frame is removed. The depth of the stack equals the longest chain of nested calls.
- `return` gives the value and ends the method at once; a statement after it cannot be reached. In a method that returns a value, every path must end in a `return`; the compiler does not reason about the values of conditions. A `void` method returns no value and can exit early with a bare `return;`.
- The returned value can widen to the return type (`int` → `double`) but cannot narrow. The return type does not change how an expression is evaluated.
- A `static` method belongs to the class; it is called directly or with `ClassName.method()`. A method that is not `static` is called through an object: `object.method()`.
- Java **passes arguments by value**. Even if the copy of a primitive argument changes, the caller's variable does not. With an array argument the reference is copied: the method can change the shared array, but assigning a new array to the parameter does not affect the caller.
- **Overloading**: same name, different parameter list. The compiler makes the choice at compile time by looking at the argument types. An exact match is looked for first; if there is none, widening is tried (`byte → short → int → long → float → double`, `char → int`). If several versions are applicable, the most specific is chosen; if there is no most specific version, the call is ambiguous.
- `System.out.println` adds a line break, `print` does not, and `printf` prints with formatting and uses the locale for the decimal separator. `+` concatenation works from left to right.
- `Math` methods are `static`. `sqrt` and `pow` always return `double`, `abs` / `max` / `min` return the type of the argument, and `round(double)` returns `long`. Packages other than `java.lang` need an `import`.
- A local variable lives in the block where it is defined and must be given a value before use. An inner scope sees the outside, and an outer scope cannot see the inside. An inner block cannot use the same name as an outer local; a local variable, however, **shadows** a field of the same name. `static` fields live for the whole program.
- Design: **abstraction** (what it does matters, how it does it is hidden), **generalization** (constants become parameters), **decomposition** (each subtask becomes a method), **DRY** (repeated code goes into a method), single responsibility, descriptive names, returning the result and leaving the printing to the caller.

3.2.14 Key terms

The key terms of this chapter are listed below.

function method method call parameter argument return type return value method signature access modifier static member call stack stack frame pass by value reference heap method overloading widening / narrowing conversion import package locale local variable field scope shadowing garbage collector abstraction decomposition

3.2.15 Test Yourself

1. The methods `static int a(int n) { return b(n + 1) * 2; }`, `static int b(int n) { return c(n) + n; }` and `static int c(int n) { return n * n; }` are given. If `a(2)` is called from `main`, what is returned? What is the greatest number of frames on the stack at the same time?

   Answer

   `a(2)` calls `b(3)`. `b(3)` calls `c(3)`, and `c(3)` returns 9. `b(3)` returns 9 + 3 = 12. `a(2)` returns 12 × 2 = 24. While `c(3)` runs, there are 4 frames on the stack, `main`, `a`, `b` and `c`; this is the deepest moment.

2. The method `static int change(int[] d, int k) { d[k] = d[k] + k; k = 0; return d[k]; }` is called in `main` with `int[] d = {3, 4, 5}; int k = 2; int r = change(d, k);`. What are `d`, `k` and `r` after the call?

   Answer

   The method's `d` is a copy of the reference to the array in `main`; its `k` is a copy of 2. `d[2] = 5 + 2 = 7` changes the shared array. `k = 0` changes only the copy. `return d[0]` returns 3. Result: the array is `{3, 4, 7}`, `k` in `main` is still 2, and `r = 3`.

3. A class has the methods `m(long x)`, `m(double x)` and `m(int x, int y)`; each returns its own parameter types as a string. After the declarations `short s = 7; char ch = 'x';`, which

versions do `m(s)`, `m(ch)`, `m(3.0f)` and `m(2, 'b')` call?

Answer

In the calls with one argument, the version with two parameters is eliminated because of the count. `short` can widen to both `long` and `double`; `long` is more specific: `m(s)` → long. `char` widens first to `int` and from there to `long`; again `long` is chosen: `m(ch)` → long. `float` widens only to `double`: double. In `m(2, 'b')`, `'b'` widens to `int`: int,int. JDK 21 output: `long long double int,int`.

4. A class has the field `static int count = 0;` and the method `static int increment(int count) { count++; TryIt.count += 2; return count; }`. After the line `int t = increment(5);`, what are `t` and the class field `count`?

Answer

The parameter `count` shadows the field. `count++` makes the parameter 6; the field is not affected. `TryIt.count += 2` reaches the field through the class name and makes it 2. The method returns the parameter: `t = 6`. The field is 2.

5. What does the method `static double ratio(int a, int b) { return a / b; }` return for `ratio(7, 2)` and `ratio(1, 3)`? How is the method fixed?

Answer

`a / b` is an integer division between two `int`s. 7 / 2 = 3 and 1 / 3 = 0; on the way out they are widened to `double`: `3.0` and `0.0`. The return type does not change the calculation. Fix: `return (double) a / b;`. The results are then 3.5 and 0.333….

6. Which of the following pairs of methods can exist together in the same class? (a) `void g(int x)` and `int g(int y)`; (b) `int h(int a, double b)` and `int h(double a, int b)`; (c) `int k(int a)` and `int k(int a, int b)`.

Answer

(a) cannot: both signatures are `g(int)`; the return type and the parameter name are not part of the signature. (b) can exist together, because the order of the parameter types differs. But a call such as `h(1, 1)` gives the error `reference to h is ambiguous`; the compiler cannot choose one of the two versions. (c) can exist together: the numbers of parameters differ.

7. A method is written like this: `static String status(int points) { if (points >= 50) return "pass"; else if (points < 50) return "fail"; }`. Does it compile? If not, what is the smallest

fix?

Answer

It does not compile: `missing return statement`. Although the two conditions cover every integer, the compiler does not evaluate the values of conditions; because the last branch of the chain is an `else if`, it assumes that control can reach the end of the method. The smallest fix is to write a plain `else` instead of `else if (points < 50)`.

8. What do the lines `System.out.println("Result: " + 2 + 3);` and `System.out.println(2 + 3 + " result");` print? What does `System.out.printf("%.1f%n", 2.25);` print on a computer set to Turkish?

Answer

`+` works from left to right. In the first line `"Result: " + 2` first becomes a string, and then 3 is appended: `Result: 23`. In the second line `2 + 3` is first added as numbers: `5 result`. Because `printf` uses the locale, the decimal separator is a comma and the number is rounded to one digit: `2,3`.

9. You want to split the job “read a student's grades from the keyboard, compute the average, and print ‘pass’ if it is greater than 50” into methods instead of a single `main`. Which methods, with which parameters and return types, could be written?

Answer

One possible design: `int[] readGrades(Scanner in, int n)` reads and returns the array; `double average(int[] grades)` computes and returns; `boolean hasPassed(double avg)` decides. The printing stays in `main`. Each method does a single job, takes its data through parameters and returns its result. `average` and `hasPassed` can be tested without touching the keyboard.

3.2.16 Further reading

- A. B. Downey and C. Mayfield, *Think Java* (2nd ed., O'Reilly, 2019), Chapter 4: explains writing methods, parameters and stack diagrams with short examples; it can be read free online.
- J. Bloch, *Effective Java* (3rd ed., Addison-Wesley, 2018), the “Methods” chapter: an experienced programmer's advice on checking parameters, designing method signatures, and when overloading causes trouble.
- R. C. Martin, *Clean Code* (Prentice Hall, 2008), Chapter 3 “Functions”: discusses the principles of short methods, doing one thing, few parameters and side-effect-free design on example code.

- J. Gosling et al., *The Java Language Specification*, Java SE 21, §15.12: defines with precise rules how the compiler finds the applicable methods for a call and how it chooses the most specific one.

3.2.17 References

- Oracle, *The Java Tutorials*: “Defining Methods”, “Passing Information to a Method or a Constructor”.
- J. Gosling et al., *The Java Language Specification*, Java SE 21 Edition (§8.4 Method Declarations, §6.3 Scope of a Declaration, §15.12 Method Invocation Expressions).
- P. Deitel and H. Deitel, *Java: How to Program, Early Objects*, 11th ed., Pearson, 2017 (Chapter 6: Methods).
- K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly, 2022.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., 2022.
- A. B. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019 (Chapter 4: Methods and Testing).
- J. Bloch, *Effective Java*, 3rd ed., Addison-Wesley, 2018.
- R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2008.

PART I · CHAPTER 3.3

Arrays

The exam grades of a class, the readings a sensor delivers every second, or the pixels of a picture all consist of many values of the same kind. Creating a separate variable for each of these values is tiring, and the code breaks when the number of values changes. The array solves this problem: it holds many values of the same type under a single name, in numbered cells. These notes first explain the array concept, the index and the layout of an array in memory. They then cover declaring arrays in Java, initializing them, `length`, the `for-each` loop and the `java.util.Arrays` class. The fact that an array variable is a reference, together with copying and passing arrays to methods, is the most important topic of this part. The following sections deal with the out-of-bounds error, statistical calculations (mean, median, mode), searching and sorting algorithms and their costs, and multidimensional arrays. The notes end with a Tic Tac Toe game built on a two-dimensional character array.

LEARNING OUTCOMES

Concept
Explain that an array holds a fixed number of elements of the same type in consecutive cells, and that elements are accessed by an index that starts at 0.

Syntax
Declare one- and multidimensional arrays; create them with `new` and with an initializer list; traverse them with `length`.

Memory
Show that an array variable is a reference and that the array lives on the heap; predict the outcome of assignment, copying and passing to a method.

Errors
Recognize `ArrayIndexOutOfBoundsException` and common compile errors; prevent them with bounds checks.

Algorithms
Write mean, median, mode, linear and binary search, matching, reversal, selection sort and bubble sort; compare the costs $O(1)$, $O(\log n)$, $O(n)$ and $O(n^2)$.

Application
Fill and print matrices with nested loops, transpose them, and build a Tic Tac Toe game with `char[][]`.

3.3.1 The array concept

In the programs so far, every variable held a single value: a number, a character or a string. In real problems, however, values often come in collections. The grades of fifty students, the temperature measured every minute through a day, or the node voltages in a circuit are such collections. This section defines the array, shows why it is needed, and explains how an array sits in memory.

3.3.1.1 What is an array?

Definition 3.3.1

An **array** is a structure that holds a fixed number of values of the same type under a single name, in consecutive cells. Each value in the array is called an **element**, and the position number of an element in the array is called its **index**. The index of the first element is 0; the last index of an array with n elements is $n - 1$.

An array can be thought of as a variable defined with one name that can hold several values. A single variable has one name and one value. An array has a single name but many values. The values are told apart by the index. To reach an element, you write the name of the array followed by the index in square brackets: `arr[0]`, `arr[1]`, `arr[2]`.

Two everyday analogies make the concept concrete. The first is an apartment building. The building has a single address; this corresponds to the name of the array. The flats are numbered, and saying "number 3" is enough to find the right flat. Nobody gives directions like "the second door from the right on the top floor". The second is an egg carton. The number of cells in the carton is known from the start, and the carton does not grow later. Only things of the same kind are put in each cell. The two basic properties of an array, fixed size and a single type, show up in this analogy.

Indexing from 0 feels odd to beginners. In everyday life counting starts at 1: the first student, the second student. Java, C and the languages derived from them see the index as an *offset*. The index says how many cells away from the start of the array the element is. The first element is 0 cells away from the start, so its index is 0. This view also makes it easy to compute the address of an element in memory: element number `i` is `i` cells away from the starting address.

Example 3.3.1

Let `arr = [10, 20, 30, 40, 50]`. The array has 5 elements, so the valid indices are 0, 1, 2, 3 and 4.

`arr[0]` is the first element and its value is 10. `arr[4]` is the last element and its value is 50. The third element is `arr[2]`; the index is one less than the ordinal number. There is no element `arr[5]`. Such an access causes an error at run time in Java.

3.3.1.2 Why arrays?

Suppose the average of the grades of fifty students is to be computed. Without an array, a separate variable is needed for each grade:

```
int grade1 = 70;
int grade2 = 85;
int grade3 = 90;
// ... up to grade50
int sum = grade1 + grade2 + grade3 /* + ... */;
```

This approach has three problems. First, fifty names have to be written one by one in the sum expression. Second, if a student joins the class, the program has to change: a new variable is created and added to the sum. Third, the names of the variables are meaningful only to the programmer. Java does not know the "ordering" relationship between `grade1` and `grade2`. A loop cannot combine `grade` and `i` to produce the name `grade3`.

The array solves these three problems together:

```
int[] grades = new int[50];
// ... the grades are filled in
int sum = 0;
for (int i = 0; i < grades.length; i++) {
    sum += grades[i];
}
```

Here `grades` is a single name and stands for fifty cells at once. The loop counts `i` from 0 to 49 and looks at one cell with `grades[i]` on each pass. The same code works for 5 grades and for 5000 grades; only the size of the array changes. Because `grades.length` in the loop condition tells how many cells the array has, the condition does not have to be changed by hand either.

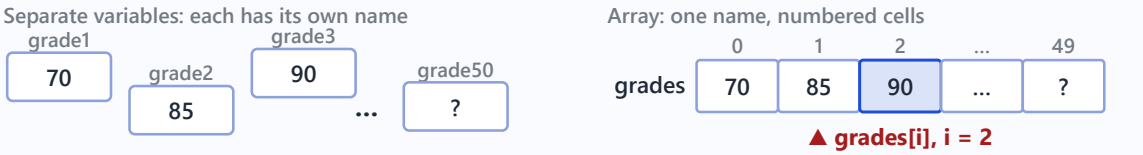


Figure 3.3.1: On the left, separate variables whose names are independent of each other; on the right, an array with a single name and numbered cells. When `grades[i]` is written, the current value of `i` determines which cell is looked at; in the figure, with `i` = 2, the value 90 is read.

The real power of the array can be seen in Figure 3.3.1. The index does not have to be a constant number; it can be a variable or an expression. When `grades[i]` is written, which cell is looked at is decided not at compile time but at run time, from the value of `i`. That is why a single line of code can be applied to all the cells in turn inside a loop.

3.3.1.3 An array in memory

The elements of an array are not scattered at random places in memory. The cells are laid out side by side, **consecutively**. Figure 3.3.2 shows an `int` array with five elements. Above each cell is its index, and inside it is its value.



Figure 3.3.2: The array `a` with five elements. The cells are next to each other in memory; the indices run from 0 to 4. The variable `a` holds not the cells themselves but a reference that points to the start of the array.

index	0	1	2	3	4
element	13	−1	4	2	7

Table 3.3.1: The indices and elements of the array `a`.

According to Table 3.3.1, `a[1]` = −1, `a[4]` = 7 and `a.length` = 5. There is no cell `a[5]`; the valid indices are 0…4. This distinction is the source of most array-related bugs. The *length* of the array is the number of cells; the *last index* is one less than the length.

The arrow in the figure matters too. The variable `a` does not contain five numbers. The variable holds a **reference** to the region of memory where the five numbers sit. This detail looks unimportant at first sight, but it determines how array assignment, copying and passing to a method work.

3.3.1.4 Advantages and disadvantages

The consecutive layout is the source of both the greatest strength and the greatest limitation of the array.

The strengths are the following. The array stores data in an orderly, consecutive way. Access by index is fast: `arr[i]` is read in the same time for every `i`. There is no difference between accessing the first element and accessing the millionth. Because the data is grouped, it is easily processed with loops. Memory use is also lean: the array stores only the elements and its length. In structures such as a linked list, the address of the next element is stored next to each element; an array has no such overhead.

The limitations come from the same layout. First, the array has a **fixed size**: its size cannot be changed after it is created. To add a new element when there is no room left, a new, larger array is created and the old elements are copied into it. Second, inserting in the middle or deleting from the middle requires shifting elements. Third, an array holds elements of only one type.

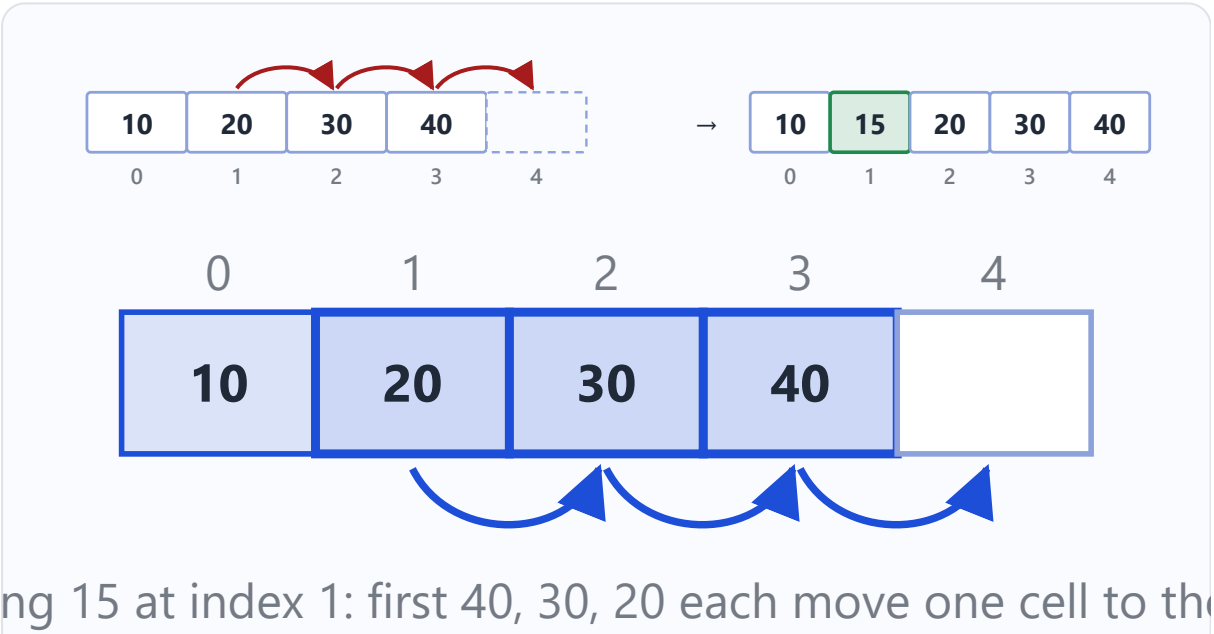


Figure 3.3.3: Inserting into the middle of an array. (a) 15 is inserted at index 1 of an array whose last cell is empty; 20, 30 and 40 move one cell to the right and the array becomes `[10, 15, 20, 30, 40]`. (b) The same shifting with arrows: each element is moved to the cell on its right.

Figure 3.3.3 shows the cost of insertion. If 15 is to be put at index 1, all elements after index 1 must be moved one cell to the right. The direction of the shifting matters. If shifting goes from left to right, 20 is first written over 30 and 30 is lost. That is why one starts from the last element:

Example 3.3.2

In the array `int[] d = {10, 20, 30, 40, 0};` the last cell is empty. Let 15 be inserted at index 1.

The loop `for (int i = 3; i >= 1; i--) d[i + 1] = d[i];` first performs `d[4] = d[3]` (40), then `d[3] = d[2]` (30), and finally `d[2] = d[1]` (20). Then `d[1] = 15;` is written.

The result is `[10, 15, 20, 30, 40]`. If there are k elements after the insertion position, k shifts are needed. In the worst case, inserting at the front, all the elements move; so insertion needs O(n) work.

For lists whose size changes while the program runs, Java has the `ArrayList` class. `ArrayList` still uses an array inside. When the array is full, it creates a larger array and copies the elements into it. The programmer does not see this work, but the cost does not go away; it is only hidden. `ArrayList` and the other collections are in the `java.util` package.

3.3.1.5 Kinds of arrays and where they are used

Arrays are divided into three groups by the number of dimensions. In a **one-dimensional** array the elements sit in a single row; `int[] grades` is of this kind. In a **two-dimensional** array the elements are arranged in rows and columns; this is a matrix and is written `int[][] table`. A **multidimensional** array is written with more square brackets; `int[][][] cube` is three-dimensional.

The array is the most widespread data structure in programming languages. Three uses often met in electrical and electronic engineering are the following:

- **Measurements.** Samples taken from a sensor every second are kept in a `double[]` array. A digitized audio signal is the same. One second of sound sampled at 44.1 kHz is an array of 44,100 elements.
- **Matrices.** Node equations in circuit analysis, systems of linear equations and transformation matrices are expressed with two-dimensional arrays.
- **Images.** A grayscale image is a grid of pixels of the form `int[height][width]`. A color image needs one more dimension: the red, green and blue channels of each pixel.

Example 3.3.3

A temperature sensor gives 1000 readings, and the readings are kept in the array `double[] readings = new double[1000];`. How many bytes do the elements take?

In Java every `double` is 8 bytes. $1000 \cdot 8 = 8000$ bytes, that is, about 8 KB is needed. If the same readings were `int`, each would take 4 bytes and the total would be 4000 bytes. The JVM sets aside a few more bytes for the length and type of the array; this extra is small next to the

elements.

This calculation is useful in design. A `double` array of one million elements takes 8 MB, and a 1000 × 1000 `double` matrix again takes 8 MB. As the number of dimensions grows, memory grows quickly.

3.3.2 Arrays in Java

The array concept is similar in all languages, but the details vary from language to language. How an array is declared in Java, what value the cells start with, where the length is read from, and what an array variable holds are the subjects of this section. The section ends with the `java.util.Arrays` class, which is often used with arrays.

3.3.2.1 Declaring arrays and default values

In Java an array is declared and created with the following general pattern:

```
Type[] arrayName = new Type[arraySize];
```

The pattern has three parts. `Type` is the type of the values to be stored in the array: `int`, `double`, `char`, `String` or any class. `arrayName` is the name of the array and obeys the rules for naming variables. `arraySize` says how many elements the array will contain. The size can be a constant, or an `int` computed at run time.

Two separate things are actually done in the line. The left side, `Type[] arrayName`, declares an array *variable*. The right side, `new Type[arraySize]`, sets aside room for the array in the region of memory called the heap. The equals sign writes the reference of this new array into the variable. The variable and the array are different things. This distinction is the basis of all the reference topics ahead.

```
int[] numbers = new int[5];      // 5 integers
double[] readings = new double[100]; // 100 decimals
String[] names = new String[30];
int n = 12;
char[] letters = new char[n];     // the size comes from a variable
int arr2[] = new int[3];         // C style, valid but not recommended
```

All of these lines compile. `numbers` has five elements, `readings` a hundred, and `names` thirty. The size of the array `letters` comes from a variable: since `n` is 12, the array has 12 elements. The notation `int arr2[]` in the last line is a leftover from C. Java accepts it, but it is recommended to write the square brackets next to the type: `int[]` reads as “int array” and the whole type appears in one place.

The size must be known at the moment of creation. It does not change afterwards. Even if the value of `n` changes after `new int[n]` is written, the size of the array stays the same. If the size is negative, the array cannot be created; the program throws `NegativeArraySizeException`:

```
Exception in thread "main" java.lang.NegativeArraySizeException: -2
        at NegativeSize.main(NegativeSize.java:4)
```

An array of size 0, on the other hand, is valid. `new int[0]` is an array with no cells and its `length` is 0. Such an array is useful for expressing "no results at all" without using `null`.

Default values

The cells of an array created with `new` do not stay empty. Java writes a **default value** into each cell according to its type.

Type	Default value
`int`, `long`, `short`, `byte`	`0`
`double`, `float`	`0.0`
`boolean`	`false`
`char`	`'\u0000'` (the character with code 0)
`String` and all object types	`null`

Table 3.3.2: Default values of array elements.

Default values are interesting when compared with local variables. If a local variable declared as `int x;` is read without being given a value, the compiler reports the error `variable x might not have been initialized`. Array cells, on the other hand, are never "valueless". `new int[5]` contains five 0s and can be read at once.

The default value of a `char` array needs separate attention. `'\u0000'` is not the space character; it is a character with code 0 that is invisible on the screen. The code of the space character is 32. A program that wants to show empty cells as spaces must first fill the `char` array with `' '`. The cells of a `String[]` array are also not the empty string (`""`) but `null`. `null` is a reference that points to no object, and if a method is called on it, a `NullPointerException` occurs.

Common mistake

After the line `String[] names = new String[30];`, `names[0].length()` is called. Because the cell is `null`, the program stops with a `NullPointerException`. `new String[30]` does not create thirty strings; it creates thirty empty references that can point to thirty strings. A string must be assigned to each cell separately.

3.3.2.2 Access and `length`

The elements of an array are accessed by index, and the index starts at 0. The expression `arr[i]` is used both to read and to write. On the right of the equals sign it reads, on the left it writes. The size of the array is taken from the `length` property.

```
int[] numbers = {10, 20, 30, 40, 50};
int first = numbers[0];    // gets the first element
int second = numbers[1]; // gets the second element
int arraySize = numbers.length; // 5
System.out.println(first + " " + second
        + " " + arraySize);
numbers[1] = 25;               // write
int last = numbers[numbers.length - 1]; // 50
```

The program prints the following line:

```
10 20 5
```

`numbers[0]` reads the first element (10) and `numbers[1]` reads the second (20). `numbers.length` is 5. The `" "` strings in `println` separate the numbers from each other; without strings between them, `first + second` would add and print 30. The line `numbers[1] = 25;` writes to the second cell. The last line is the general pattern for reading the last element: `numbers[numbers.length - 1]`. However many elements the array has, this expression gives the last element (an empty array has no last element).

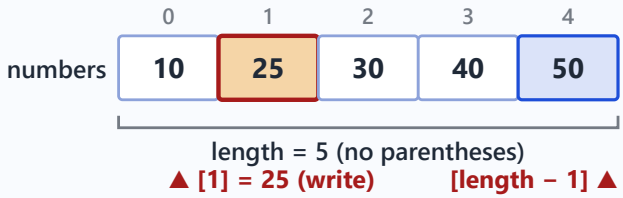


Figure 3.3.4: The array `numbers` after the write. `numbers[1] = 25` has changed the second cell. `length` = 5 is the number of cells; the last cell is `[length - 1]`, that is, `[4]`.

`length` is not a method but a property (a field) of the array. That is why it is written without parentheses: `numbers.length`. `length()` in the `String` class, on the other hand, is a method and is written with parentheses. The two notations are easily confused:

```
E_LengthParen.java:4: error: cannot find symbol
        System.out.println(a.length());
                           ^
  symbol:   method length()
  location: variable a of type int[]
1 error
```

The message says that no method named `length()` was found in the variable `a` of type `int[]`. The reverse is wrong too: if `s.length` is written for a `String` variable, the compiler cannot find `variable length`. A short rule can be kept in mind: no parentheses for arrays, parentheses for strings.

The `length` field of an array is `final`, that is, it cannot be changed. The line `numbers.length = 10;` does not compile. The fixed size of the array is also guaranteed by this rule.

3.3.2.3 Initialization

If the elements of the array are known in advance, there is no need to assign each cell one by one. The values are given in braces, separated by commas. This notation is called an **array initializer**.

```
int[] numbers = {10, 20, 30, 40, 50};      // with 5 elements
int[] n = {1, 2, 3, 4, 5};   // there are 5 values, so n is an array of 5 elements
int[] m = new int[]{1, 2, 3}; // long form: can be used anywhere
String[] days = {"Mon", "Tue", "Wed"};
char[] vowels = {'a', 'e', 'ı', 'i', 'o', 'ö', 'u', 'ü'};
```

When an initializer list is used, the size is not written. The number of values in the list determines the size: the array `n` has five values, so `n.length` is 5. `days` has three elements and `vowels` eight. Because Java stores characters as Unicode, Turkish letters such as `'ı'`, `'ö'` and `'ü'` are also valid `char` values.

The initializer list has two forms. The short form, `{1, 2, 3}`, can be used **only on the declaration line**. The long form, `new int[]{1, 2, 3}`, is valid everywhere: in an assignment, in a method call and in `return`. If you try to declare the array first and fill it later with the short form, the compiler reports an error:

```
E_ShortAfter.java:4: error: illegal start of expression
        a = {1, 2, 3};
            ^
E_ShortAfter.java:4: error: not a statement
        a = {1, 2, 3};
             ^
E_ShortAfter.java:4: error: ';' expected
        a = {1, 2, 3};
              ^
3 errors
```

Of the three errors in the message, the real one is the first: `illegal start of expression`. After the `=` sign the compiler expects an expression and does not recognize the brace as the start of an expression. The next two messages are confusion caused by this first error. After one error the compiler often produces a series of extra messages; once the first message is fixed, the others disappear too. The correct form is `a = new int[]{1, 2, 3};`.

Giving both the size and the list is also forbidden. Since the list already determines the size, Java does not allow the possibility that the two pieces of information contradict each other:

```
E_BothGiven.java:3: error: array creation with both dimension expression and
initialization is illegal
        int[] a = new int[3]{1, 2, 3};
                             ^
1 error
```

Giving neither is an error too. If `new int[]` is written and no list follows, the compiler cannot know how many cells the array will have:

```
E_NoSize.java:3: error: array dimension missing
        int[] a = new int[];
                          ^
1 error
```

The values in the list must fit the type of the array. `2.5` cannot be put in an `int` array, because converting a `double` to an `int` discards the fractional part and Java does not do this on its own:

```
E_Lossy.java:3: error: incompatible types: possible lossy conversion from double to int
        int[] a = {1, 2.5, 3};
                      ^
1 error
```

The opposite direction causes no problem. `double[] d = {1, 2, 3};` is valid; the `int` values widen to `double` without loss and the array holds `1.0, 2.0, 3.0`. The list `double[] c = {-40, 0, 25, 100};` compiles for the same reason.

> **Common mistake**
>
> Out of habit from C, one writes `int a[3] = {1, 2, 3};`. In Java the size is not part of the type of the variable; javac gives the message `']' expected`. In Java the type is only "int array"; how many cells there are is a property of the object, not of the variable. The correct form is `int[] a = {1, 2, 3};`.

3.3.2.4 Code trace: ArrayExample

The following program declares an array, assigns values to its cells one by one, reads two elements, and finally prints all the elements with a loop. Although it is short, it contains all the basic operations on an array.

```
public class ArrayExample {
    public static void main(String[] args) {
        // Declaring an array: an integer array with 5 elements
        int[] numbers = new int[5];
        // Assigning values to the elements of the array
        numbers[0] = 10;
        numbers[1] = 20;
        numbers[2] = 30;
        numbers[3] = 40;
        numbers[4] = 50;
        // Accessing the elements of the array
        System.out.println("first element: " + numbers[0]);
        System.out.println("third element: " + numbers[2]);
        // Printing all the elements of the array in a loop
        System.out.print("Elements: ");
        for (int i = 0; i < numbers.length; i++) {
            System.out.print(numbers[i] + " ");
        }
    }
}
```

The output of the program:

```
first element: 10
third element: 30
Elements: 10 20 30 40 50
```

`new int[5]` creates an array of five cells on the heap. Each cell starts with the default value 0. The variable `numbers` holds the reference to this array. Then five assignment lines fill the cells in turn. The index of the first element is 0, so `numbers[0]` reads 10. The third element is `numbers[2]`; the index is one less than the ordinal number.

In the last part `print` is used. Unlike `println`, `print` does not move to the next line; so all the elements are written on the same line with a space between them. A space is also left at the end of the line. The loop starts `i` at 0 and continues as long as the condition `i < numbers.length` is true.

`i`	`i < numbers.length`	Cell read	Appended to the screen
0	0 < 5 true	`numbers[0]` = 10	`10`
1	1 < 5 true	`numbers[1]` = 20	`20`
2	2 < 5 true	`numbers[2]` = 30	`30`
3	3 < 5 true	`numbers[3]` = 40	`40`
4	4 < 5 true	`numbers[4]` = 50	`50`
5	5 < 5 false	—	the loop ends

Table 3.3.3: Trace of the printing loop in the ArrayExample program.

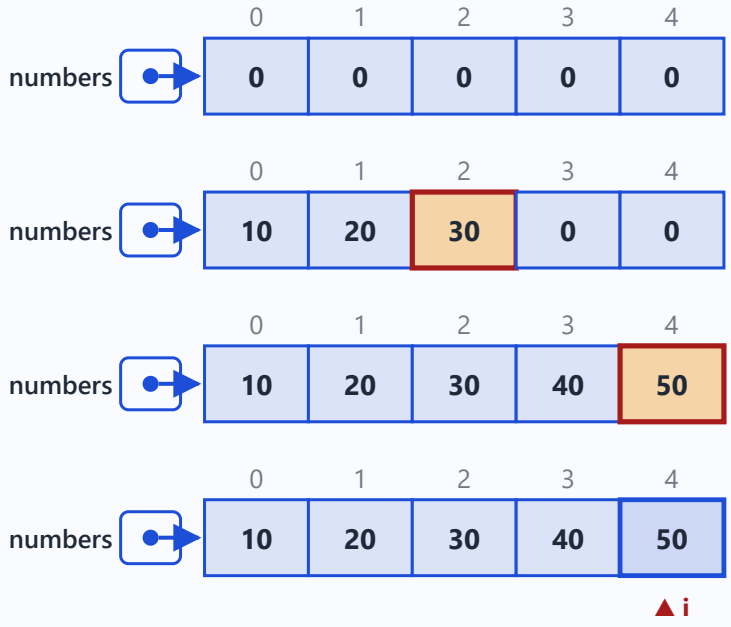


Figure 3.3.5: Four moments of the ArrayExample trace. (a) After `new int[5]`: all five cells are 0. (b) After three assignments: 10, 20, 30. (c) All five assignments are done. (d) In the printing loop with `i` = 4: the last valid cell is read. On the next pass `i` becomes 5 and the condition turns false.

Table 3.3.3 shows that the loop checks the condition six times while the body runs five times. At the last check `i` is 5 and 5 < 5 is false. The last valid index is `length - 1` = 4.

The pattern `for (int i = 0; i < arr.length; i++)` traverses all indices of an array from 0 to `length - 1`. This pattern is used so often that it is written from memory. Two details of the pattern matter: the start is 0 and the comparison is `<`. Writing `<=` is a common mistake and goes beyond the array on the last pass.

The cost of the loop is clear as well. The body runs once per element; in an array with *n* elements there are *n* passes. The amount of work grows in proportion to the size of the array. This cost is called O(n).

3.3.2.5 The `for-each` loop

Processing all the elements of an array in order is so common that Java offers a separate loop for it. This loop is called the **enhanced** `for` or, in short, the `for-each` **loop**.

> **Definition 3.3.2**
>
> The `for-each` **loop** of the form `for (type element : arr) { … }` puts a copy of the next element of the array into the variable `element` on each pass and runs the body. The loop makes as many passes as there are elements in the array. No index is written.

The colon makes it easier to read. The colon is read as “for each … in”: `for (int grade : grades)` means “for each `grade` in `grades`”. The type of the loop variable must be compatible with the element type of the array. A `double[]` array cannot be traversed with an `int` variable, because each pass would need a narrowing conversion:

```
E_ForEachType.java:4: error: incompatible types: possible lossy conversion from double to
int
        for (int x : d) {
                     ^
1 error
```

The following program traverses the same array first with a classic `for` and then with `for-each`:

```
int[] grades = {70, 85, 90};
// Classic for: we have the index
for (int i = 0; i < grades.length; i++) {
    System.out.println(i + ". grade: " + grades[i]);
}
// for-each: only the values
int sum = 0;
for (int grade : grades) {
    sum += grade;
}
System.out.println("Sum: " + sum);
```

```
0. grade: 70
1. grade: 85
2. grade: 90
Sum: 245
```

In the classic `for` loop we have the index; this is how the ordinal information `0.`, `1.`, `2.` in the output gets written. In the `for-each` loop there is no index; only the values come. Since the sum needs no index, the second loop is shorter and less prone to errors.

The `for-each` loop has three properties. First, since no index is written, an out-of-bounds error cannot be made; the loop itself knows where the array ends. Second, the loop is only for reading. Writing `grade = 0;` does not change the element in the array, it changes only the copy. Third, it is not suitable for jobs that need the index. Writing to the array, reporting the position of an element, or comparing two neighboring elements requires an index. In these cases the classic `for` is used.

3.3.2.6 Code trace: the value comes, not the cell

The following program shows most clearly that the `for-each` loop is for reading:

```
int[] scores = {60, 75, 95};
int sum = 0;
for (int p : scores) {
    sum += p;
}
System.out.println("Sum: " + sum);
for (int p : scores) {
    p = 0;
}
System.out.println(scores[0] + " " + scores[2]);
```

The program prints `Sum: 230` and then `60 95`. The first loop adds up the three scores. The second loop tries to zero each element, but the array does not change.

Loop	Pass	`p`	Body	`sum`	`scores`
—	start	—	—	0	[60, 75, 95]
1	1	60	`sum += p`	60	[60, 75, 95]
1	2	75	`sum += p`	135	[60, 75, 95]
1	3	95	`sum += p`	230	[60, 75, 95]
2	1	60 → 0	`p = 0`	230	[60, 75, 95]
2	2	75 → 0	`p = 0`	230	[60, 75, 95]
2	3	95 → 0	`p = 0`	230	[60, 75, 95]

Table 3.3.4: Trace of two `for-each` loops. `p` is a copy of the next element on each pass.

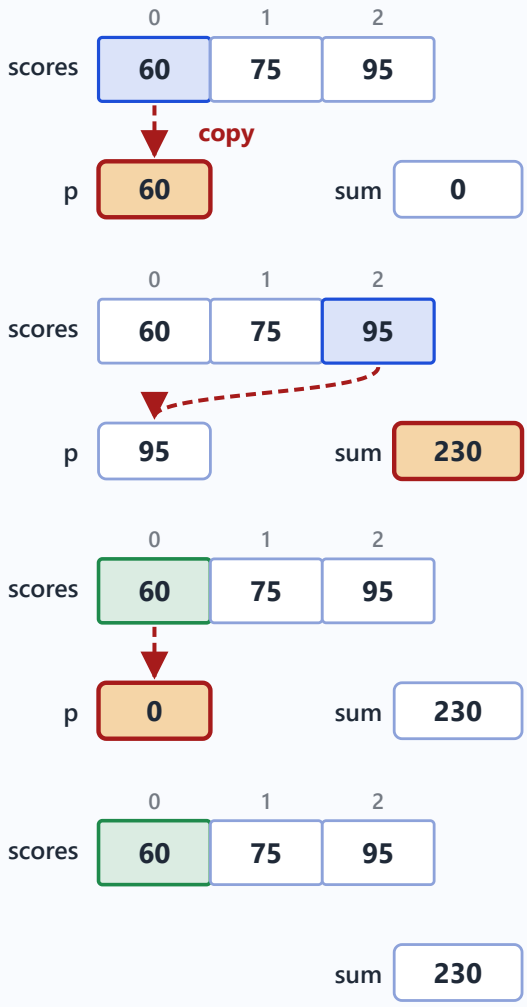


Figure 3.3.6: Four moments of the `for-each` trace. (a) First pass: the copy of `scores[0]` goes to `p`, `p` = 60. (b) At the end of the last pass `sum` = 230. (c) In the second loop `p = 0` changes only the copy; the 60 in the array stays in place. (d) When the loops are finished the array has not changed at all.

Figure 3.3.6 gives the reason visually. `p` is not a cell of the array but a separate local variable. At the start of each pass the loop copies the value of the next cell into `p`. `p = 0` writes to this local variable; no link remains between the cell and `p`. To write to the array, a classic `for` and `scores[i] = 0` are needed.

Example 3.3.4

Let every element of the array `int[] x = {5, 6, 7};` be multiplied by two.

The wrong way: `for (int v : x) v = v * 2;`. On the passes `v` becomes 10, 12 and 14 in turn, but these values are written to copies. `Arrays.toString(x)` still gives `[5, 6, 7]`.

The right way: `for (int i = 0; i < x.length; i++) x[i] = x[i] * 2;`. This time the cells themselves are written and the array becomes `[10, 12, 14]`.

3.3.2.7 Temperature conversion

Arrays and the `for-each` loop together give a short way to produce a table. The following program converts four Celsius values to Fahrenheit and Kelvin:

```
double[] celsiusValues = {-40, 0, 25, 100};
for (double celsius : celsiusValues) {
    double fahrenheit = (celsius * 1.8) + 32;
    double kelvin = celsius + 273.15;
    System.out.println(celsius + " °C = " + fahrenheit
            + " °F = " + kelvin + " K");
}
```

```
-40.0 °C = -40.0 °F = 233.14999999999998 K
0.0 °C = 32.0 °F = 273.15 K
25.0 °C = 77.0 °F = 298.15 K
100.0 °C = 212.0 °F = 373.15 K
```



Figure 3.3.7: The array `celsiusValues`. The values written as `int` widen to `double` and are stored as -40.0, 0.0, 25.0, 100.0.

`double[] celsiusValues = {…};` gathers four values in one array. The list is written with `int` constants, but because the array is `double`, the values are stored as `-40.0`, `0.0`, `25.0` and `100.0`. The `.0`s in the output come from this. `for (double celsius : celsiusValues)` means "for each element in the array". The body runs four times and the table is produced in one go.

Pass	`celsius`	`celsius * 1.8 + 32`	`celsius + 273.15`
1	-40.0	-72.0 + 32 = -40.0	233.14999999999998
2	0.0	0.0 + 32 = 32.0	273.15
3	25.0	45.0 + 32 = 77.0	298.15
4	100.0	180.0 + 32 = 212.0	373.15

Table 3.3.5: Trace of the temperature conversion.

The `233.14999999999998` in the first row looks like a bug, but Java works correctly. Floating-point numbers are stored in binary. Many decimal numbers such as 273.15 have no finite representation in binary. The number is stored as the nearest representable value, and a small rounding difference shows up in the result of the operation. For the same reason `0.1 + 0.2` gives `0.30000000000000004`.

This difference can be hidden by limiting the number of digits when printing: `System.out.printf("%.2f K%n", kelvin)`. Here there is a detail specific to Turkey. `printf` formats the number according to the language setting of the computer. On a computer set to Turkish the output is `233,15 K`; the decimal separator is a comma. `println`, on the other hand, always uses a point, independent of the setting.

Common mistake

Comparing floating-point numbers with `==`. The expression `-40 + 273.15 == 233.15` gives `false`. The equality of two floating-point numbers is checked by testing whether the difference between them is less than a small threshold: `Math.abs(a - b) < 1e-9`.

3.3.2.8 An array variable is a reference

This is the most important and most misunderstood property of arrays. When `int x = 5;` is written, the number 5 sits inside the variable `x`. When `int[] a = {1, 2, 3};` is written, however, the three numbers do not sit inside `a`. The three numbers are in an array on the heap; `a` holds the address of that array, that is, its reference.

Definition 3.3.3

A **reference** is a value that points to the location of an object in memory. Arrays, strings and all class objects are **reference types**: variables of these types hold not the object itself but a reference to the object. If two variables point to the same object, one is an **alias** of the other.

```
int[] a = {1, 2, 3};
int[] b = a;         // not a copy!
b[0] = 99;
System.out.println(a[0]);
int[] c = a.clone(); // a real copy
c[1] = 77;
System.out.println(a[1] + " " + c[1]);
System.out.println(a == b);
System.out.println(a == c);
```

The program prints four lines: `99`, `2 77`, `true` and `false`. The reason for each line can be seen in Figure 3.3.8.

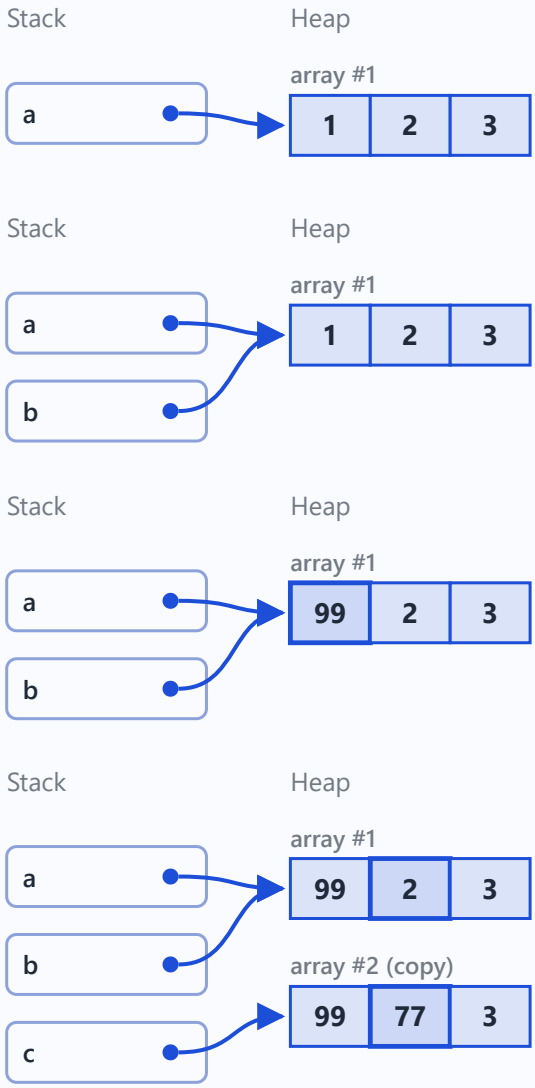


Figure 3.3.8: Trace of array references. (a) An array is created on the heap; `a` holds its reference. (b) `b = a` copies only the reference; both variables point to the same array. (c) Because `b[0] = 99` changes the shared

array, `a[0]` also becomes 99. (d) `clone()` creates a new array on the heap; `c[1] = 77` changes only this copy.

Let us follow the lines in order. `int[] b = a;` does not create a new array. The value of `a` on the right-hand side is a reference, and this reference is copied into `b`. Now both variables point to the same array. `b[0] = 99` writes to the first cell of the shared array. When `a[0]` is read, the same cell is looked at, so 99 comes out.

`a.clone()`, on the other hand, produces a real copy. A new array is created on the heap and the elements are copied into it. Because `a[0]` is 99 at the moment of copying, the copy is `[99, 2, 3]`. `c[1] = 77` changes only the copy; in the original array `a[1]` is still 2.

The last two lines show the meaning of the `==` operator on arrays. `==` compares references, that is, addresses. `a == b` is true because both point to the same array. `a == c` is false. Even if the array in `c` holds the same contents as that of `a` at that moment, it is a different object. To compare contents, `Arrays.equals` is used.

Passing an array to a method

In Java all arguments are **passed by value**: a copy of the argument goes to the method. When an `int` is passed, a copy of the number goes. When an array is passed, a copy of the *reference* goes. The copied reference also points to the same array. So if the method changes the cells of the array, the change is visible in the caller's array.

Example 3.3.5

Let the following method and call be given:

```
static void modify(int[] d, int x) { d[0] = 100; x = 100; }
int[] a = {1, 2, 3}; int x = 1; modify(a, x);
```

After the call `a[0]` is 100 and `x` is 1. `d` is a copy of the reference in `a` and points to the same array; `d[0] = 100` writes to a cell of that array. The `x` in the method is a copy of the `x` in `main`. Writing 100 to the copy does not affect the original variable.

This rule has one more consequence. If a new array is assigned to the parameter inside the method, for example `d = new int[]{0, 0};`, only the local copy is bound to the new array. The caller's variable keeps pointing to the old array. Whatever the method writes to `d` from then on goes to the new array, and the caller does not see it.

Common mistake

Mistaking assignment for copying. If `int[] backup = grades;` is written and `grades` is sorted afterwards, `backup` is sorted too, because there is only a single array. For a real copy, `grades.clone()` or `Arrays.copyOf(grades, grades.length)` is used.

3.3.2.9 The `java.util.Arrays` class

Java offers ready-made methods for common jobs with arrays. These methods are in the `java.util.Arrays` class, and to use them `import java.util.Arrays;` is written at the top of the file. All the methods are `static`; they are called with the class name, as in `Arrays.sort(d)`.

```
int[] arr = {5, 3, 8, 1};
System.out.println(Arrays.toString(arr));
Arrays.sort(arr);
System.out.println(Arrays.toString(arr));
int[] copy = Arrays.copyOf(arr, 6);
System.out.println(Arrays.toString(copy));
int[] filled = new int[4];
Arrays.fill(filled, 7);
System.out.println(Arrays.toString(filled));
int[] x = {1, 2};
int[] y = {1, 2};
System.out.println((x == y) + " " + Arrays.equals(x, y));
System.out.println(Arrays.binarySearch(arr, 5));
```

```
[5, 3, 8, 1]
[1, 3, 5, 8]
[1, 3, 5, 8, 0, 0]
[7, 7, 7, 7]
false true
2
```

Line by line: `Arrays.toString` turns the array into readable text. `Arrays.sort` sorts the array from smallest to largest. The sorting is done **in place**; no new array is returned, the given array changes. `Arrays.copyOf(arr, 6)` creates a new array with 6 cells and copies the old elements into the first four cells; the extra cells get the default value 0. If a new length smaller than the old one is given, the array is cut: `copyOf(arr, 2)` gives `[1, 3]`.

`Arrays.fill(filled, 7)` writes the same value to all the cells. `x` and `y` are two separate arrays holding the same elements: `x == y` is false and `Arrays.equals(x, y)` is true. `Arrays.binarySearch(arr, 5)` on the last line searches for 5 in the sorted array and returns its index: 2.

Method	Purpose
`Arrays.toString(d)`	Turns the array into text in the form `[5, 3, 8, 1]`.
`Arrays.deepToString(m)`	Turns nested arrays (matrices) into text.
`Arrays.sort(d)`	Sorts the array in place, from smallest to largest.
`Arrays.binarySearch(d, x)`	Returns the index of `x` in a sorted array; if it is absent, a negative number is returned.
`Arrays.copyOf(d, n)`	Creates a new array with `n` cells and copies into it; extra cells get the default value.
`Arrays.fill(d, v)`	Writes `v` to all the cells.
`Arrays.equals(a, b)`	Compares the lengths and the elements of two arrays.

Table 3.3.6: Frequently used methods of the `java.util.Arrays` class.

Printing an array directly with `println` does not give the expected result. `System.out.println(arr)` prints not the elements but an identifier such as `[I@1c20c684`. The leading `[I` means “int array”; the hexadecimal number after `@` is the identity code of the object and may change on every run. To see the elements, `Arrays.toString(arr)` is used.

Two more details should be known. First, arrays have no search method such as `indexOf`. To search an unsorted array a loop is written; this method is called linear search. Second, `binarySearch` works correctly only on a sorted array. If the sought value is absent, the result is a negative number that also encodes the position where the value would be inserted. If 4 is searched for in the sorted array `[1, 3, 5, 8]`, the result is −3. On an unsorted array the result is meaningless, and no warning is given either.

The costs of the methods differ. `toString`, `fill`, `copyOf` and `equals` touch each element once, so they are O(n). `binarySearch` is O(log n), and `sort` is O(n log n) on average. The one-line form of `sort` does not mean that the work behind it is small.

3.3.3 Memory and errors

The most frequent run-time error in programs that use arrays is going outside the bounds of the array. To understand this error you first need to know where an array lives in memory. This section explains how Java divides memory, how unused arrays are cleaned up, and how bounds errors are caught.

3.3.3.1 Where are arrays in memory?

The memory of a running Java program is divided into two main regions. The **stack** holds the local variables of method calls. Every method call opens a **frame** on the stack; when the method ends, the frame is removed and the variables in it disappear. The **heap** is the region where objects created with `new` live. An object on the heap can live on after the method that created it has ended.

In Java all arrays are on the heap. Only the reference variables that point to the arrays sit on the stack. In C and C++ the situation is different. An array declared as `int a[10];` sits on the stack. A raw array allocated with `malloc` or `new[]` is on the heap, and when the programmer is done with it, it must be released by hand with `free` or `delete[]` (in C++, `std::vector` does this by itself). In Java there is no such distinction; releasing is automatic too.

```
String[] plates = {"", "Adana", "Adıyaman", "Afyon"};
System.out.println(plates[1]);
int[] dynamicArr = new int[10]; // space is allocated on the heap
System.out.println(dynamicArr.length);
dynamicArr = null; // reference cut
System.out.println(dynamicArr == null);
```

The program prints `Adana`, `10` and `true` in turn. The first cell of the `plates` array is the empty string; so the province with plate code 1 sits at `plates[1]`, and the index and the plate code coincide. This is a practical compromise with indexing from 0. The size of `dynamicArr` could also have been given at run time; but once the array is created, its size is fixed.

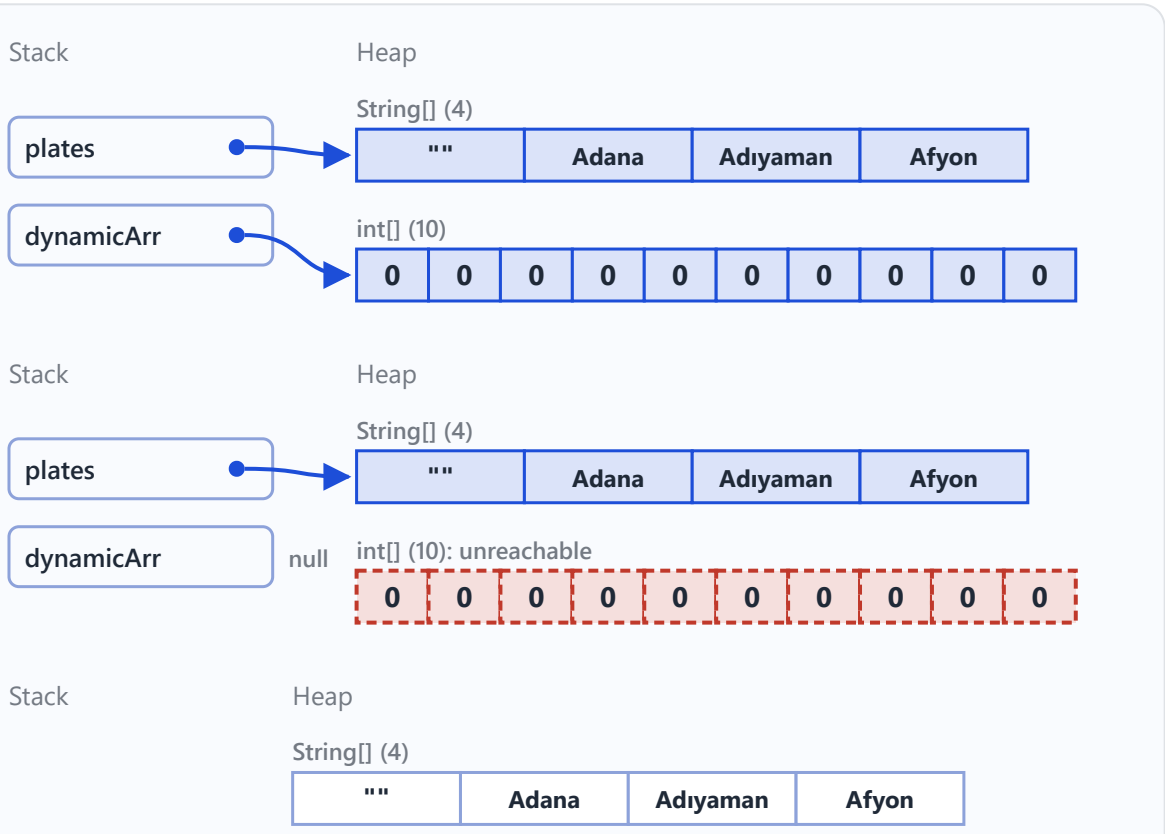


Figure 3.3.9: Arrays and the garbage collector. (a) Both arrays are on the heap; only reference variables sit on the stack. The cells of the `String[]` are in fact references to separate `String` objects. (b) `dynamicArr = null` cuts the link; the `int` array becomes unreachable. (c) When the method ends the stack frame is removed; arrays with no other reference are collected in time.

The line `dynamicArr = null;` does not delete the array. It only cuts the link between the variable and the array. As seen in Figure 3.3.9, the array stays on the heap, but there is no longer any reference that reaches it. The JVM's **garbage collector** finds such unreachable objects and reclaims their memory.

Definition 3.3.4

The **garbage collector** is the JVM component that finds objects that no reference points to and gives back the memory they occupy. Java has no release command such as `free` or `delete`; memory management is automatic.

It is not defined when the garbage collector will run. An array becoming unreachable does not mean that the memory will be freed at once. The JVM does the collecting at a suitable moment. What matters for the programmer is not to hold on to unneeded references. If a variable that points to a large array lives until the end of the program, the array lives too.

Example 3.3.6

What happens to the first array after the lines `int[] p = new int[1000]; int[] q = p; p = new int[5]; q = null;`?

In the first line an array of 1000 cells is created and `p` points to it. In the second line `q` points to the same array too. The third line binds `p` to a new array of 5 cells; the first array is still pointed to by `q`. The fourth line cuts this last link as well.

Now no reference reaches the array of 1000 cells, and the garbage collector can collect it. The array has not shrunk: `p = new int[5]` creates a new array and does not change the old one.

3.3.3.2 The index-out-of-bounds error

The valid indices of an array lie between 0 and `length - 1`. If an array is accessed with an index outside this range, Java throws an `ArrayIndexOutOfBoundsException` at run time. This exception is a subclass of `IndexOutOfBoundsException`. If the exception is not caught, the program stops at that line.

Definition 3.3.5

`ArrayIndexOutOfBoundsException` is the exception the JVM throws when an array is accessed with a negative index or with an index equal to or greater than `length`. The compiler does not check index values; the check is made at run time on every access.

This behavior sets Java apart from C. In C an access such as `a[10]` is not checked. The program reads or writes the memory right behind the array. The result is a random value, the corruption of another variable, or an obscure bug that shows up much later. Java, on the other hand, catches the error on the very line where it occurs and stops the program. The program crashes, but it does not silently produce a wrong result.

```
int[] arr = {10, 20, 30, 40, 50};
int index = 5;
// Checking the array length
if (index >= 0 && index < arr.length) {
    int value = arr[index];
    System.out.println("Value: " + value);
} else {
    System.out.println("Invalid index: " + index);
}
// Unchecked access:
System.out.println(arr[index]);
```

```
Invalid index: 5
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds
for length 5
        at OutOfBounds.main(OutOfBounds.java:13)
```

The first part of the program shows the pattern for safe access: before the index is used, it is checked with the condition `index >= 0 && index < arr.length`. Because 5 is not a valid index, the `else` branch runs and `Invalid index: 5` is printed. The last line uses the same index without a check, and the exception occurs.

The exception message says everything needed to find the error. `Index 5` is the faulty index. `out of bounds for length 5` is the length of the array. The line `at OutOfBounds.main(OutOfBounds.java:13)` gives the class, the method and the line number of the file where the error occurred. When you meet a bounds error, this line is the first place to look.

A negative index causes the same exception. In some languages `d[-1]` means the last element; in Java it does not:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index -1 out of
bounds for length 3
        at Negative.main(Negative.java:4)
```

Common mistake

Thinking the last index is `length`. `a[a.length]` is always out of bounds; the last element is `a[a.length - 1]`. The same error appears in a loop condition as `i <= a.length`. For an array with three elements the message is `Index 3 out of bounds for length 3`: the faulty index and the length are equal.

3.3.3.3 The classic bug: a loop with `<=`

The most common source of the out-of-bounds error is the loop condition:

```
int[] arr = new int[3];
for (int i = 0; i <= arr.length; i++) {
    arr[i] = i * 10;
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds
for length 3
        at BoundsLoop.main(BoundsLoop.java:5)
```



Figure 3.3.10: When `i` = 3, the condition 3 <= 3 stays true and an attempt is made to write to the fourth cell that does not exist, `arr[3]`.

The loop writes 0, 10 and 20 to the cells for `i` = 0, 1, 2. When `i` becomes 3, `3 <= 3` is true and the body runs once more. `arr[3]` does not exist and the exception is thrown. The correct condition is `i < arr.length`. Writing a constant, for example `i < 3`, gives the right result for this array; but if the size of the array changes, the code breaks. Using `length` ties the condition to the array.

To see in which pass the error occurs, let us trace a similar program step by step:

```
int[] square = new int[4];
for (int i = 0; i <= square.length; i++) {
    square[i] = i * i;
}
System.out.println("done");
```

`i`	`i <= square.length`	**Body**	`square`
—	—	start	[0, 0, 0, 0]
0	0 <= 4 true	`square[0] = 0`	[0, 0, 0, 0]
1	1 <= 4 true	`square[1] = 1`	[0, 1, 0, 0]
2	2 <= 4 true	`square[2] = 4`	[0, 1, 4, 0]
3	3 <= 4 true	`square[3] = 9`	[0, 1, 4, 9]
4	4 <= 4 true	`square[4]` does not exist: exception	the program stops

Table 3.3.7: Trace of a loop written with `<=`; `square.length` = 4.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds
for length 4
        at Square.main(Square.java:5)
```

Table 3.3.7 shows the exact place of the error. The last valid cell is filled on the fourth pass. On the fifth pass `4 <= 4` is also true; had `<` been written, the loop would end here. The JVM throws the exception at the access `square[4]` and the program stops on this line. `"done"` is never printed, because the exception is not caught and so the following lines are not reached.

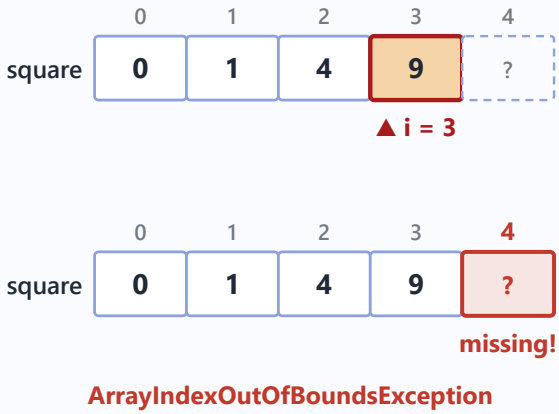


Figure 3.3.11: (a) `i` = 3: the last valid cell is filled with `square[3] = 9`. (b) `i` = 4: `square[4]` does not exist; the JVM throws `ArrayIndexOutOfBoundsException`.

The root of the error is the habit of counting. Someone who counts from 1 to 4 writes `<=`; someone who starts at 0 and counts four elements must write `< 4`. When the two patterns are mixed up, either the first element is skipped or the last pass goes beyond the array. `for (int i = 1; i <= d.length; i++)` does both: `d[0]` is never written, and the last pass throws the exception.

3.3.3.4 An array that was never created: `null`

If an array variable is declared and no array is bound to it, or if `null` is assigned to it explicitly, the variable points to no array. Accessing a cell through such a variable causes a `NullPointerException`:

```
Exception in thread "main" java.lang.NullPointerException: Cannot load from int array
because "<local1>" is null
        at Npe.main(Npe.java:4)
```

Since Java 15 these messages are "helpful" by default: they also say what is `null`. The `"<local1>"` in the message is local variable number 1 of the method, that is, `a`. By default the compiler does not write local variable names into the `.class` file. If the code is compiled with `javac -g`, or if debugging information is added as in most IDEs, the message becomes `because "a" is null` directly.

The difference between an array and `null` is the difference between an empty array and absence. `int[] a = new int[0];` is an array and `a.length` gives 0. `int[] a = null;` is not an array; even `a.length` throws a `NullPointerException`.

3.3.4 Statistics examples

The three best-known measures for summarizing a data set are the mean, the median and the mode. All three are computed naturally with arrays, and each shows a different technique. The mean adds up in a single pass, the median relies on sorting, and the mode compares neighbors in a sorted array. All three programs start with the same section that reads the data from the user.

3.3.4.1 Reading an array from the user

```
import java.util.Scanner;
public class AverageCalculation {
    public static void main(String[] args) {
        // Get the size of the array from the user
        Scanner scanner = new Scanner(System.in);
        System.out.print("enter size: ");
        int size = scanner.nextInt();
        // Get the elements from the user and build the array
        double[] arr = new double[size];
        for (int i = 0; i < size; i++) {
            System.out.print((i + 1) + ". element: ");
            arr[i] = scanner.nextDouble();
        }
        // ... calculations with the array
    }
}
```

In a sample run the user first enters the size and then four values:

```
enter size: 4
1. element: 7
2. element: 3
3. element: 9
4. element: 5
```

In this program the size of the array is not known at compile time. `scanner.nextInt()` reads the number the user typed, and `new double[size]` creates the array with that size. The size is decided at run time, but it does not change once the array has been created. If the user wants to enter more values, a new array is needed.

The `(i + 1)` in the prompt line is a small but important detail. The user counts from 1 and expects to see “1. element”. The code counts from 0 and writes the value to `arr[i]`. `i + 1` is used on the screen and `i` in the code. The parentheses are also needed: `(i + 1) + ". element: "` does the addition first. Without parentheses, `i + 1 + "."` would be evaluated from left to right and still give the right result; but a form such as `"" + i + 1` would produce `"01"`. The parentheses show the intention clearly.

The `nextDouble()` method of `Scanner` reads the decimal separator according to the language setting of the computer. On a computer set to Turkish, `7,5` is accepted and `7.5` causes an `InputMismatchException`. With integer input this problem does not arise.

3.3.4.2 Mean

The **mean** (arithmetic mean) is the sum of the values divided by the number of values. After the reading section, the program runs two more loops: one adds up, the other prints.

```
Scanner scanner = new Scanner(System.in);
System.out.print("enter size: ");
int size = scanner.nextInt();
double[] arr = new double[size];
for (int i = 0; i < size; i++) {
    System.out.print((i + 1) + ". element: ");
    arr[i] = scanner.nextDouble();
}
double sum = 0;
for (int i = 0; i < size; i++) {
    sum += arr[i];
}
double mean = sum / size;
System.out.print("Elements: ");
for (int i = 0; i < size; i++) {
    System.out.print(arr[i] + " ");
}
System.out.println();
System.out.println("Mean: " + mean);
```

```
enter size: 4
1. element: 7
2. element: 3
3. element: 9
4. element: 5
Elements: 7.0 3.0 9.0 5.0
Mean: 6.0
```

Step	`i`	Operation	`arr`	`sum`
reading	—	`size = 4`, `new double[4]`	[0.0, 0.0, 0.0, 0.0]	—
reading	0	`arr[0] = 7.0`	[7.0, 0.0, 0.0, 0.0]	—
reading	1	`arr[1] = 3.0`	[7.0, 3.0, 0.0, 0.0]	—
reading	2	`arr[2] = 9.0`	[7.0, 3.0, 9.0, 0.0]	—
reading	3	`arr[3] = 5.0`	[7.0, 3.0, 9.0, 5.0]	—
summing	—	`sum = 0`	same	0.0
summing	0	`sum += arr[0]`	same	7.0
summing	1	`sum += arr[1]`	same	10.0
summing	2	`sum += arr[2]`	same	19.0
summing	3	`sum += arr[3]`	same	24.0
division	—	`mean = 24.0 / 4`	same	6.0

Table 3.3.8: Trace of AverageCalculation; the user enters 4, then 7, 3, 9, 5.

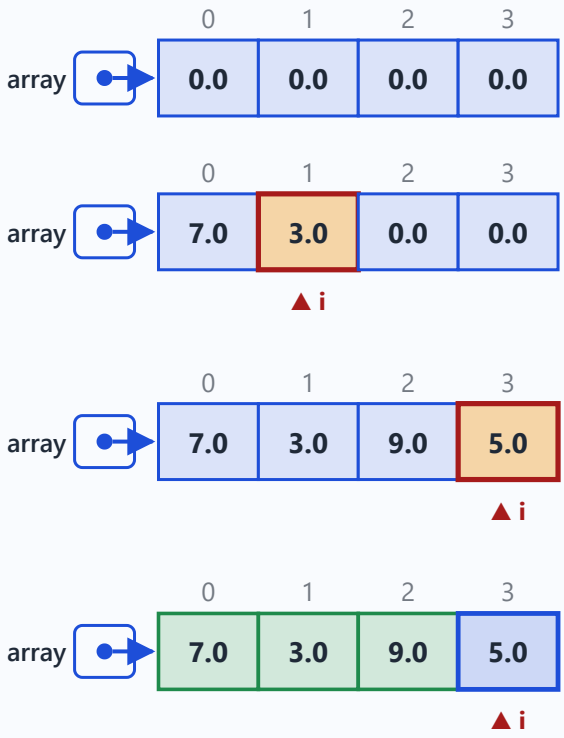


Figure 3.3.12: Four moments of the mean trace. (a) `new double[size]` : all four cells are 0.0. (b) Two values have been read. (c) Four values have been read: 7.0, 3.0, 9.0, 5.0. (d) The last pass of the summing loop: `sum` = 24.0.

The floating-point division is the result of a design decision. Because `sum` is declared `double` , `sum / size` is the quotient of a `double` and an `int` . The `int` is first widened to `double` and real division is performed: 24.0 / 4 = 6.0. If the sum were `int` , the difference would show. For the set 7, 3, 9, 5, 1, 4, 29 / 6 gives 4 with integer division; the correct value is 4.833….

Common mistake

Computing the mean with two `int` s and assigning the result to a `double` : `double mean = sum / arr.length;` . The division is performed *before* the assignment, between two `int` s, and the fraction has already been discarded; 29 / 6 first becomes 4 and then becomes 4.0. The right way is to convert before dividing: `sum / (double) arr.length` .

The cost of the program is clear. The reading, summing and printing loops each make n passes; the total work is of the order $3n$, which is O(n). The sum could also have been accumulated in the reading loop; merging the two loops reduces the number of passes but does not change how the cost grows.

3.3.4.3 Median

Definition 3.3.6

The **median** is the value in the middle of a sorted data set. If the number of elements is odd, it is the element exactly in the middle; if even, it is the mean of the two middle elements.

The median is not affected by extreme values as much as the mean is. The mean of the values 3, 4, 5, 6 and 100 is 23.6 and is close to none of the data. Its median is 5. That is why the median is preferred for measurements that contain a few extreme values, such as income, house prices or network latency.

```
// size and array are read as in AverageCalculation
Arrays.sort(arr); // Sort the array
double median;
if (size % 2 == 0) { // Even number of elements
    int mid1 = size / 2 - 1;
    int mid2 = size / 2;
    median = (arr[mid1] + arr[mid2]) / 2;
} else { // Odd number of elements
    median = arr[size / 2];
}
System.out.println("Median: " + median);
```

```
entered: 7 3 9 5 1
Sorted elements: 1.0 3.0 5.0 7.0 9.0
Median: 5.0
```

The first step of the calculation is sorting. The median is defined on the sorted order; the element in the middle of an unsorted array is a random value. Figure 3.3.13 shows this: the cell in the middle in the order entered is 9.0, but the median is 5.0.

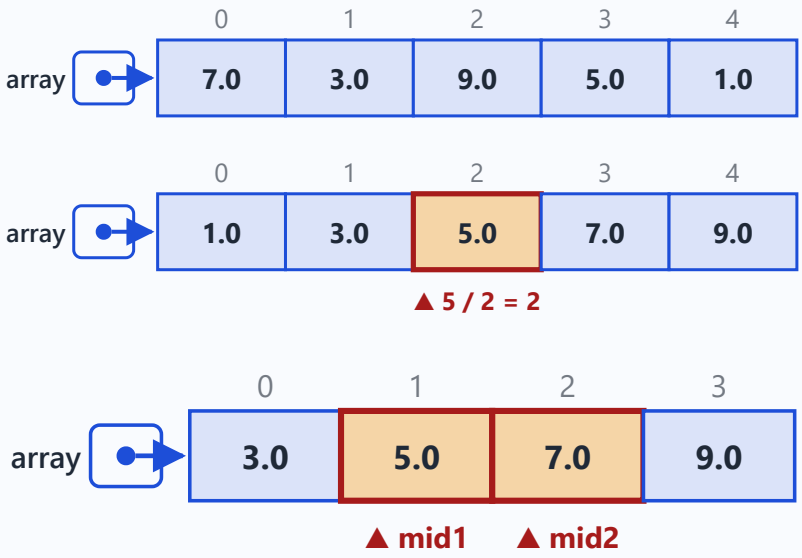


Figure 3.3.13: The median. (a) The entered values are unsorted; the cell in the middle (9.0) is not the median. (b) After `Arrays.sort`, size 5 (odd): `arr[5 / 2]` = `arr[2]` = 5.0. (c) Size 4 (even; inputs 7 3 9 5): `mid1` = 1, `mid2` = 2 and the median is (5.0 + 7.0) / 2 = 6.0.

If the size is odd, the middle index is `size / 2`. For size 5, 5 / 2 gives 2 with integer division, not 2.5. Here the fact that integer division discards the fraction is useful. With indices starting at 0, the middle of 5 elements really is at index 2: there are two elements on its left and two on its right.

If the size is even, there is no single middle element. For size 4 the two middle indices are `4 / 2 - 1` = 1 and `4 / 2` = 2. If the inputs are 7, 3, 9, 5, the sorted array is `[3.0, 5.0, 7.0, 9.0]` and the median is (5.0 + 7.0) / 2 = 6.0. Because the array is `double`, the division is floating-point. If the same code were written with an `int` array, an expression such as (4 + 5) / 2 would give 4; the right value is 4.5, and `/ 2.0` must be written.

The line `double median;` in the code is outside the branches. This placement is necessary. If the variable were declared inside the `if` and `else` branches, each would live only in its own block. The `println` outside the branches could not see it:

```
E_Median.java:10: error: cannot find symbol
        System.out.println("Median: " + median);
                                       ^
  symbol:   variable median
  location: class E_Median
1 error
```

Declaring the variable outside and giving it a value in only one branch is not enough either. Without the `else` branch the compiler would see that the variable stays without a value when the `if` condition is false and would report `variable median might not have been initialized`. In the form used in the code, both branches assign a value, so there is no problem.

`Arrays.sort` has a side effect: it changes the array in place. After the median is computed, the order entered by the user is lost. If the order is needed later, the sorting is done on a copy: `double[] sorted = arr.clone(); Arrays.sort(sorted);`.

The cost is determined by the sorting. Reading the middle element is O(1), but `Arrays.sort` does O(n log n) work. The median is a more expensive measure than the mean.

3.3.4.4 Mode

Definition 3.3.7

The **mode** is the value that occurs most often in a data set. The mode of the set 4 2 7 2 4 2 9 is 2; 2 occurs three times.

A plain way to find the mode is to count, for each value, how many times it occurs by scanning the whole array. This way makes *n* comparisons for each of the *n* values and costs $O(n^2)$. The program in the slides takes a smarter way. It first sorts the array, so that equal values come next to each other. Then in a single pass it finds the longest "run".

```
int modeValue = arr[0];
int modeCount = 1;
int currentValue = arr[0];
int currentCount = 1;
for (int i = 1; i < size; i++) {
    if (arr[i] == arr[i - 1]) {
        currentCount++;
    } else {
        currentCount = 1;
        currentValue = arr[i];
    }
    if (currentCount > modeCount) {
        modeCount = currentCount;
        modeValue = currentValue;
    }
}
```

Four variables keep track of two runs. `currentValue` and `currentCount` are the value and the length of the run being counted at the moment. `modeValue` and `modeCount` are the value and the length of the longest run seen so far. The loop starts from `i` = 1, because each element is compared with the one before it, and the first element has no predecessor. The first element is initially counted as both the current run and the longest run.

Two decisions are made on every pass. First: if the element equals the previous one, the run grows; if not, a new run starts and the length becomes 1. Second: if the current run is longer than the longest run, the longest run is updated. The result is printed like this:

```
// Print the result
System.out.println("Sorted elements: " + Arrays.toString(arr));
System.out.print("Mode: " + modeValue);
System.out.println(" (count: " + modeCount + ")");
```

```
Sorted elements: [2, 2, 2, 4, 4, 7, 9]
Mode: 2 (count: 3)
```

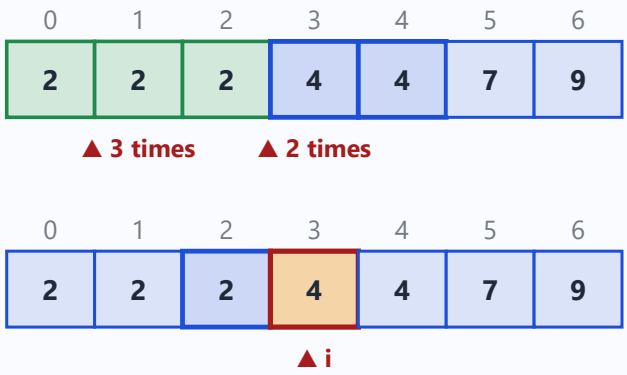


Figure 3.3.14: The mode in a sorted array. (a) The runs: 2 three times, 4 twice, 7 and 9 once each. (b) `i` = 3: `arr[3]` = 4 differs from the previous one (2) and a new run starts.

`i`	Comparison	Current run	Longest run
—	start	2 × 1	2 × 1
1	2 == 2	2 × 2	2 × 2 (updated)
2	2 == 2	2 × 3	2 × 3 (updated)
3	4 ≠ 2	4 × 1	2 × 3
4	4 == 4	4 × 2	2 × 3
5	7 ≠ 4	7 × 1	2 × 3
6	9 ≠ 7	9 × 1	2 × 3
7	7 < 7 false: the loop ends. Mode 2, count 3.		

Table 3.3.9: Trace of the mode loop; sorted array `[2, 2, 2, 4, 4, 7, 9]`.

The `>` sign in the update condition decides the case of a tie. If two runs have the same length, the run that comes later cannot overtake the earlier one; the first one found is kept. Because the array is sorted, this means that the smallest of the equally frequent values is chosen. Had `>=` been written, the last one found, that is, the largest, would have been chosen. If it is necessary to report that there may be several modes, additional counting is needed.

Example 3.3.7

What is the mode of the sorted array `{2, 2, 6, 6, 6, 9}` ?

The first run is 2 × 2; the longest run is 2 × 2. At `i` = 2, 6 starts (6 × 1). At `i` = 3 the run becomes 6 × 2, but 2 > 2 is false and the longest run stays 2. At `i` = 4 the run becomes 6 × 3. 3 > 2 is true and the longest run is updated. 9 starts a new run.

The program prints `Mode: 6 (count: 3)` .

The cost of this method has two parts: O(n log n) sorting and O(n) scanning. The total is O(n log n), which is much better than the $O(n^2)$ cost of the plain counting method. Sorting here is a preparation step; the calculation that follows relies on equal values standing next to each other.

3.3.5 Array algorithms

The most frequent jobs on an array are searching, comparing, sorting and reordering. This section gives the basic algorithms for these jobs and counts how much work each of them does. The counting leads to the O notation, the common language for comparing algorithms.

3.3.5.1 Linear search

Definition 3.3.8

Linear search is a search method that compares the sought value with every element of the array in turn, from the start to the end. When the value is found the search ends; if the end of the array is reached, the value is not there.

```
static boolean linearSearch(int[] arr, int target) {
    for (int element : arr) {
        if (element == target) {
            return true;
        }
    }
    return false;
}

// inside main:
int[] arr = {10, 20, 30, 40, 50};
boolean result = linearSearch(arr, 30);
```

The `linearSearch` method takes two parameters: the array to search and the target value. The `for-each` loop puts the elements into `element` in turn. If an element equals the target, `return true` ends both the loop and the method at once. If the loop ends without finding any equality, the last line is reached and `false` is returned. Because the search needs no index, `for-each` is suitable here.

Pass	`element`	`element == target`	Result
1	10	10 == 30 false	next element
2	20	20 == 30 false	next element
3	30	30 == 30 true	`return true`

Table 3.3.10: Trace of the call `linearSearch(arr, 30)`.



Figure 3.3.15: Linear search. (a) First pass: `element` = 10, not the target. (b) Third pass: 30 is found and the method returns `true`. 40 and 50 are never looked at.

When `main` calls `linearSearch(arr, 30)`, the reference to the array and the value 30 go to the method. The array is not copied. Three comparisons are enough to find the result; the method frame is removed from the stack and `result` becomes `true`. If the target were 35, all five comparisons would turn out false and the method would return `false`.

The same search can be done without writing a method. This time `break` is used to leave the loop:

```
int[] arr = {10, 20, 30, 40, 50};
// Searching for a particular element in an array
int key = 30;
boolean found = false;
for (int element : arr) {
    if (element == key) {
        found = true;
        break;
    }
} // This operation has O(n) complexity.
```

`break` leaves the loop the moment the target is found. In this example only three elements are looked at and `found` becomes `true`. Without `break` the result would still be correct, but the loop would needlessly look at 40 and 50 as well. It is important that `found` starts as `false`: if no equality is found, the right answer is "absent".

The answer to "is it there?" alone may not be enough as a search result. If you need to know *where* the element is, the index is returned and this time a classic `for` is used: `for (int i = 0; i < d.length; i++) if (d[i] == target) return i; return -1;`. Since −1 cannot be a valid index, it is used to mean "absent". If 5 is searched for in the array `{12, 7, 30, 5, 18}` the result is 3, and if 99 is searched for it is −1.

3.3.5.2 The cost of linear search

Measuring the speed of an algorithm in seconds is misleading; the same code takes less time on a fast computer. Instead, the basic operations that the algorithm performs are counted. In linear search the basic operation is a comparison.



Figure 3.3.16: Linear search for 30 makes three comparisons; 40 and 50 are not looked at. In the worst case, when the target is last or absent altogether, $n = 5$ comparisons are needed.

The number of comparisons depends on where the target is. In the **best case** the target is the first element and one comparison is enough. In the **worst case** the target is at the end of the array or absent; in an array with n elements, n comparisons are made. If the target is at a random place in the array, the average number of comparisons is about $n/2$.

Definition 3.3.9

Time complexity describes how the amount of work an algorithm does grows with the input size n. **Big-O notation** gives the shape of this growth; it drops constant factors and smaller terms. If the amount of work grows in proportion to n, the algorithm is O(n).

The time complexity of linear search is O(n). The difference between $n/2$ and n is unimportant in big-O notation: both double when the array doubles. On an unsorted array nothing better can be done. "Absent" cannot be answered without looking at every element, because an element that was not looked at might be the target.

3.3.5.3 Binary search

If the array is sorted, the situation changes. When looking up a name in a phone book, nobody starts from the first page. The book is opened in the middle; if the name sought comes before the one on that page, one moves to the left half, and if it comes after, to the right half. At each look the search space is cut in half.

Definition 3.3.10

Binary search is a method that looks at the middle element of a sorted array and continues the search in the left half if the target is smaller and in the right half if it is larger. The search space halves at every step. It works only on a sorted array.

```
static int binarySearch(int[] arr, int target) {
    int lo = 0, hi = arr.length - 1;
    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        if (arr[mid] == target) {
            return mid;          // found
        } else if (target < arr[mid]) {
            hi = mid - 1;        // left half
        } else {
            lo = mid + 1;        // right half
        }
    }
    return -1;                   // search space exhausted
}
// inside main:
int[] arr = {3, 9, 14, 27, 42, 48, 55, 61, 70};
System.out.println(binarySearch(arr, 48));
System.out.println(binarySearch(arr, 50));
```

```
5
-1
```

The method keeps the search space in two variables: `lo` is the first and `hi` the last index of the space. At the start the space is the whole array. On each pass the middle index is found with `(lo + hi) / 2`; integer division discards the fraction. If the middle element is the target, its index is returned. If the target is smaller, the right half is eliminated and `hi = mid - 1`. If the target is larger, the left half is eliminated and `lo = mid + 1`. When `lo` passes `hi`, the space is empty and −1 is returned.

Target	Pass	`lo`	`hi`	`mid`	`arr[mid]`	Decision
48	1	0	8	4	42	48 > 42: `lo = 5`
48	2	5	8	6	55	48 < 55: `hi = 5`
48	3	5	5	5	48	found: returns 5
50	1	0	8	4	42	50 > 42: `lo = 5`
50	2	5	8	6	55	50 < 55: `hi = 5`
50	3	5	5	5	48	50 > 48: `lo = 6`
50	—	6	5	—	—	6 <= 5 false: returns −1

Table 3.3.11: Binary search in the array `{3, 9, 14, 27, 42, 48, 55, 61, 70}`.

Table 3.3.11 gives the two searches side by side. For 48 the middle indices are 4 (42), 6 (55) and 5 (48). Three comparisons are enough. For 50 the same three steps are taken, then `lo` becomes 6 and `hi` becomes 5. The space is empty, and it is settled that 50 is not in the array. Linear search would have reached the same result only with nine comparisons.

The size of the gain shows as the array grows. Because the space of n elements halves at every step, it shrinks to a single element in at most about $\log_2 n$ steps. The time complexity of binary search is therefore O(log n). For n = 1000 at most 10 comparisons are needed. For n = 1,000,000 this number is at most 20. Linear search could make 1000 and 1,000,000 comparisons for the same arrays.

This has a price: the array must be sorted. Sorting takes O(n log n) work and is not worth doing for a single search. If many searches will be made on the same array, it pays to sort once and then do each search in O(log n). `Arrays.binarySearch` uses the same method; on an unsorted array its result is meaningless.

Common mistake

Writing the loop condition as `lo < hi`. In that case, when a single element is left in the space (`lo == hi`), the loop ends without looking at that element. Exactly this happens in the third pass of the search for 48. The method returns −1 although 48 is in the array. The correct

condition is `lo <= hi`. In the same way, writing `hi = mid` is also wrong: the space might not shrink and the loop would run forever.

3.3.5.4 Matching

To find the common elements of two arrays, each element is compared with every element of the other array. This job is done with two nested loops:

```
static void match(int[] arr1, int[] arr2) {
    for (int element1 : arr1) {
        for (int element2 : arr2) {
            if (element1 == element2) {
                System.out.println("Match: " + element1);
            }
        }
    }
}
```

The call `match(new int[]{3, 7, 1}, new int[]{5, 1, 3, 8})` prints `Match: 3` and `Match: 1`. In the call the arrays are created directly with the long form `new int[]{…}`, without being assigned to variables. The short form `{3, 7, 1}` cannot be used here.



Figure 3.3.17: The comparison table of matching. Each row is an element of the outer loop and each column is an element of the inner loop. Each of the 12 cells is a comparison. Two cells hold an equality.

While the outer loop has `element1` = 3, the inner loop compares it with all four elements of `arr2`. When a match is found it is printed, but the inner loop does not stop; it looks at the remaining elements too. The same work is repeated for `element1` = 7 and 1. Figure 3.3.17 shows all the comparisons as a table. A total of 3 × 4 = 12 comparisons are made.

If the arrays have n and m elements, $n \cdot m$ comparisons are needed. If both arrays have n elements, this is n^2, and matching costs $O(n^2)$. Nested loops are usually the sign of this quadratic growth. There is a faster way too: the second array is sorted first and each element of the first array is searched for with binary search. The cost of this way is $O(n \log n)$.

3.3.5.5 Complexity: O(1), O(n), O(n²)

Before comparing the costs of algorithms, the cost of the most basic operation, access by index, must be understood. `arr[2]` and `arr[1000000]` are read in the same time. The reason for this is the layout of the array in memory.



Figure 3.3.18: Memory addresses of array elements. The array starts at address 1000 and each `int` takes 4 bytes. The address of `arr[2]` is found with a single calculation: 1000 + 2 × 4 = 1008.

Because the cells are consecutive and of equal size, the address of element number `i` is computed with one multiplication and one addition:

> address(`arr[i]`) = start + `i` × element size

For `int` the element size is 4 bytes. If the start address is 1000, `arr[2]` sits at 1008 and `arr[1000000]` at 4,001,000. Both calculations take the same time; the processor does not need to look at the earlier elements. The cost of access is independent of the size of the array, and this is written O(1), that is, **constant time**. One reason why the index starts at 0 is this formula: if the index started at 1, one more subtraction would be needed on every access.

Operation	Complexity	for $n = 1000$
Access `arr[i]`	O(1), constant	1
Binary search (sorted array)	O(log n), logarithmic	≤ 10
Linear search	O(n), linear	≤ 1000
Reversal	O(n), linear	500 swaps
Matching (nested)	$O(n^2)$, quadratic	1.000.000
Selection sort	$O(n^2)$, quadratic	≈ 500,000 (exactly 499,500)

Table 3.3.12: Complexity of array operations and the approximate number of operations for $n = 1000$.

Table 3.3.12 shows the size of the differences. $O(n^2)$ algorithms are inefficient on large data sets. If n grows tenfold, the running time of an O(n) algorithm grows tenfold and that of an $O(n^2)$ algorithm grows a hundredfold. A quadratic algorithm that sorts 1000 elements in one second takes about three hours for 100,000 elements. O(1) access, on the other hand, is not affected by the input size at all.

3.3.5.6 Selection sort

Definition 3.3.11

Selection sort is a sorting method that, on each pass, finds the smallest element in the unsorted part of the array and moves it to the start of this part. After each pass the sorted part grows by one element.

```
static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int minIdx = i;
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }
        int tmp = arr[minIdx];
        arr[minIdx] = arr[i];
        arr[i] = tmp;
    }
}
```

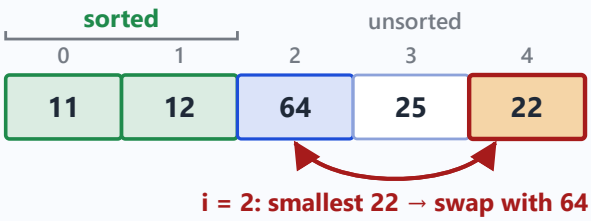


Figure 3.3.19: One pass of selection sort. The first two cells are sorted. In the pass `i` = 2 the smallest element of the unsorted part, 22, is found and exchanged with the 64 at the start of that part.

The outer loop counts `i` from 0 to `length - 2`. `i` is the start of the unsorted part. On each pass `minIdx` first becomes `i`; that is, the first element of the part is taken to be the smallest. The inner loop `j` scans the rest of the part and updates `minIdx` whenever it sees a smaller element. When the inner loop ends, the smallest element is at `arr[minIdx]` and is exchanged with `arr[i]`.

The swap is done with three assignments, with the help of a temporary variable. If `arr[i] = arr[minIdx]; arr[minIdx] = arr[i];` were written, the first assignment would erase the old value of `arr[i]` and both cells would hold the same value. `tmp` keeps the old value. It is enough for the outer loop to make `length - 1` passes: once $n - 1$ elements are in place, the last element is automatically in the right place.

The sorting is done on the array itself and the method is `void`. Because the reference to the array comes into the method, the swaps change the caller's array. This point sets a trap for those who want to move the swap into a separate method.

Common mistake

Writing the swap as `static void swap(int x, int y) { int tmp = x; x = y; y = tmp; }` **and calling** `swap(arr[i], arr[minIdx])`**.** Only copies of the elements go to the method; the method changes the copies and the array stays the same. The working version takes the array itself and the two indices: `swap(int[] d, int i, int j)`. This method exchanges `d[i]` and `d[j]`, and the change is visible in the caller's array.

In the example in the slides the array `{64, 25, 12, 22, 11}` is sorted. If a line that prints `Arrays.toString(arr)` at the end of each pass is added to `selectionSort`, the passes become visible:

```
int[] arr = {64, 25, 12, 22, 11};
selectionSort(arr);
// selectionSort prints at the end of every pass:
// System.out.println("i=" + i + ": "
//         + Arrays.toString(arr));
```

Pass `i`	Scanned `j`	Smallest (index)	Swap	Array at the end of the pass
start	—	—	—	[64, 25, 12, 22, 11]
0	1…4	11 (4)	`arr[0]` ↔ `arr[4]`	[**11**, 25, 12, 22, 64]
1	2…4	12 (2)	`arr[1]` ↔ `arr[2]`	[**11, 12**, 25, 22, 64]
2	3…4	22 (3)	`arr[2]` ↔ `arr[3]`	[**11, 12, 22**, 25, 64]
3	4…4	25 (3)	with itself	[**11, 12, 22, 25**, 64]

Table 3.3.13: Sorting the array `{64, 25, 12, 22, 11}` with selection sort.

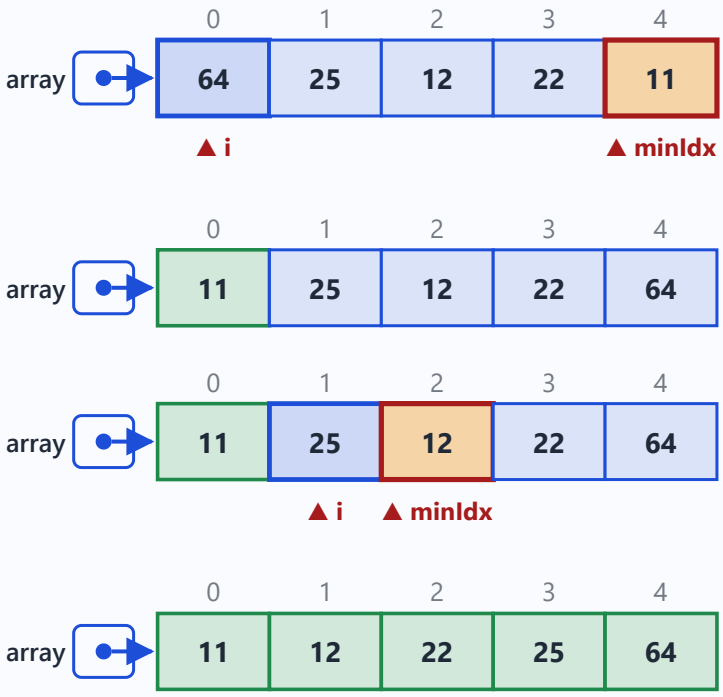


Figure 3.3.20: Trace of selection sort. (a) `i` = 0: the inner loop looks for the smallest value and finds 11 (index 4). (b) After the swap the first cell is in its final place. (c) `i` = 1: the smallest is 12 (index 2). (d) After four passes the array is sorted.

On the last pass the smallest element is already in place and is swapped with itself; the array does not change. The code does not check for this case separately. An `if (minIdx != i)` condition could avoid the needless swap, but it does not change the cost of the algorithm.

The number of comparisons is independent of the order of the input. In the first pass the inner loop looks at $n - 1$ elements, in the second at $n - 2$; in the last pass it looks at a single element. The total:

> $(n - 1) + (n - 2) + \ldots + 1 = n(n - 1) / 2$

For an array of 5 elements, $4 + 3 + 2 + 1 = 10$ comparisons are made. For $n = 1000$ it is 499,500. Even if the array is already sorted, the inner loop runs to the end on every pass; this is why selection sort is $O(n^2)$ in every case. The number of swaps, however, is at most $n - 1$. In situations where a swap is expensive this is an advantage.

3.3.5.7 Bubble sort

Definition 3.3.12

Bubble sort is a sorting method that compares two neighboring elements of the array and exchanges them if they are in the wrong order. On each pass the largest element of the remaining part rises toward the end “like a bubble”.

```
static void bubbleSort(int[] arr) {
    int n = arr.length;
    for (int pass = 0; pass < n - 1; pass++) {
        boolean swapped = false;
        for (int j = 0; j < n - 1 - pass; j++) {
            if (arr[j] > arr[j + 1]) {  // wrong order
                int tmp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = tmp;
                swapped = true;
            }
        }
        System.out.println(Arrays.toString(arr));
        if (!swapped) break;  // no swap: sorted
    }
}
```

For the array `{64, 25, 12, 22, 11, 36, 48}` the array looks like this at the end of each pass:

```
[25, 12, 22, 11, 36, 48, 64]
[12, 22, 11, 25, 36, 48, 64]
[12, 11, 22, 25, 36, 48, 64]
[11, 12, 22, 25, 36, 48, 64]
[11, 12, 22, 25, 36, 48, 64]
```

The inner loop starts `j` at 0 and compares `arr[j]` with `arr[j + 1]`. If the left one is larger, a swap is made. When the first pass is over, the largest element, 64, is at the far right; the first line of the output shows this. In the second pass the largest of the remaining six elements, 48, settles in the second cell from the right; but since 36 and 48 are already in place, this pass does not change them.

After each pass one more element at the end settles in its final place. That is why the bound of the inner loop is `n - 1 - pass`: no comparisons are made again with the elements already in place. `j + 1` must also stay inside the array. If the bound were written `n - pass`, `j` would be 6 on the first pass and an attempt would be made to access `arr[7]`.

The `swapped` flag is the clever part of the algorithm. If no swap was made in a pass, all neighboring pairs are in the right order, that is, the array is sorted. In the example the array is sorted after the fourth pass. The fifth pass makes no swap and the loop is left with `break`; there is no need for a sixth pass. This is why the last two lines of the output are the same.

Pass	Comparison	Settled	Array at the end of the pass
0	6	64	[25, 12, 22, 11, 36, 48, 64]
1	5	48	[12, 22, 11, 25, 36, 48, 64]
2	4	36	[12, 11, 22, 25, 36, 48, 64]
3	3	25	[11, 12, 22, 25, 36, 48, 64]
4	2	no swap: `break`	[11, 12, 22, 25, 36, 48, 64]

Table 3.3.14: Pass summary of bubble sort; $n = 7$.

In this run a total of 6 + 5 + 4 + 3 + 2 = 20 comparisons and 11 swaps are made. If the array were sorted from the start, no swap would occur in the first pass and the loop would end after one pass: $n - 1$ comparisons, that is, O(n). In the worst case, when the array is in reverse order, all the passes run and the number of comparisons is again $n(n - 1)/2$; $O(n^2)$. The difference from selection sort is that the number of swaps can also be $O(n^2)$. In a reverse-ordered array every comparison ends with a swap.

Example 3.3.8

In how many passes and with how many comparisons is the array `{2, 1, 3, 4, 5}` sorted by bubble sort?

In the first pass 2 and 1 are swapped and there are no swaps in the next three comparisons: 4 comparisons, array `[1, 2, 3, 4, 5]`. Because `swapped` is true, the second pass begins. In the second pass 3 comparisons are made, no swap occurs and `break` runs.

Result: 2 passes, 4 + 3 = 7 comparisons. Selection sort would make 10 comparisons for the same array.

3.3.5.8 Reversing an array

The easy but wasteful way to reverse an array is to open a second array of the same size and copy the elements into it from the end to the start. A better way is to reverse the array in place:

```java
static void reverse(int[] arr) {
    int left = 0;
    int right = arr.length - 1;
    while (left < right) {
        // Swap the start and end elements
        int tmp = arr[left];
        arr[left] = arr[right];
        arr[right] = tmp;
        // Increase the start index, decrease the end index
        left++;
        right--;
    }
}
```

```
[1, 2, 3, 4, 5]
→ [5, 4, 3, 2, 1]
[1, 2, 3, 4]
→ [4, 3, 2, 1]
```



Figure 3.3.21: The two-pointer technique. `left` comes from the left and `right` from the right; on each pass the two elements at the ends are swapped. In an array of odd length the middle element stays in place.

This method is called the **two-pointer** technique. `left` starts at 0 and `right` at the last index. On each pass the elements at the two ends are swapped, and then the pointers come one step closer to each other. When the condition `left < right` turns false, that is, when the pointers meet or pass each other, the job is done.

`left`	`right`	Condition	`tmp`	Array
0	4	0 < 4 true	1	[5, 2, 3, 4, 1]
1	3	1 < 3 true	2	[5, 4, 3, 2, 1]
2	2	2 < 2 false	—	the loop ends

Table 3.3.15: Trace of `reverse`; array `[1, 2, 3, 4, 5]`.

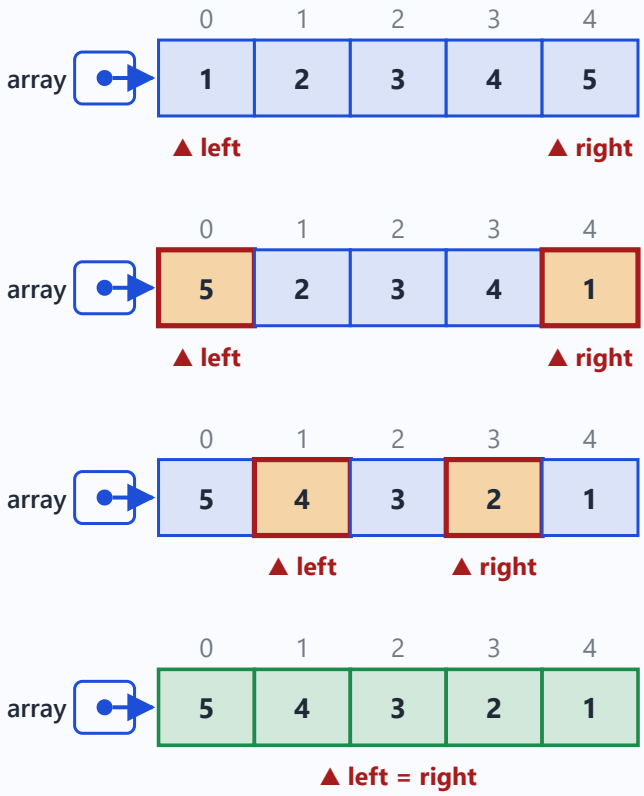


Figure 3.3.22: Trace of `reverse`. (a) The pointers at the ends. (b) `arr[0]` and `arr[4]` have been swapped. (c) `arr[1]` and `arr[3]` have been swapped. (d) `left` = `right` = 2: the loop ends and the 3 in the middle stays in place.

Two swaps are enough for an array of five elements. In an array of four elements two swaps are made as well: first 0 with 3, then 1 with 2; then `left` becomes 2 and `right` becomes 1 and the condition turns false. In general $n/2$ swaps (with integer division) are needed. Each swap is three assignments and the amount of work is proportional to n: O(n). Since no extra array is needed, the extra memory is O(1). Algorithms that work like this are called **in-place** algorithms.

The method is `void` and returns nothing. Even so, the caller's array changes, because the reference to the array came into the method. The line `→ [5, 4, 3, 2, 1]` in the output is produced by `main` printing its own array after the method call.

Common mistake

Running the reversal loop over the whole array: swapping `d[i]` and `d[d.length - 1 - i]` with `for (int i = 0; i < d.length; i++)`. In the first half the array is reversed; in the second half the same pairs are swapped again and the array returns to its old state. For `{1, 2, 3, 4, 5}` the result is again `[1, 2, 3, 4, 5]`. The loop must go over only half the array: `i < d.length / 2`.

3.3.6 Multidimensional arrays

Some data is naturally in the form of a table: the grades of a class by course, the admittance matrix of a circuit, the pixel grid of a picture. It is possible to keep such data in a one-dimensional array, but the row and column information then has to be managed by hand with index arithmetic. Multidimensional arrays leave this job to the language. Java supports arrays of two or more dimensions.

3.3.6.1 Two-dimensional arrays

Definition 3.3.13

A **two-dimensional array** is a table-like array whose elements are arranged in rows and columns. An element is accessed with `matrix[row][col]` : the first index gives the row and the second gives the column.

```
int[][] matrix = new int[3][3]; // a matrix of size 3x3
matrix[1][2] = 6;  // row 2, column 3
```

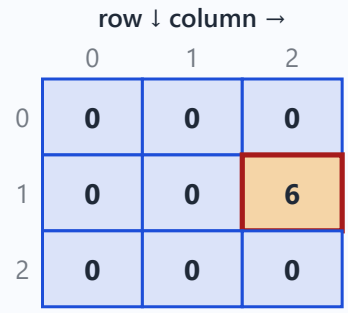


Figure 3.3.23: A 3×3 matrix. All cells start at 0; `matrix[1][2] = 6` writes to the third column of the second row.

`new int[3][3]` creates a matrix of 3 rows and 3 columns. As with one-dimensional arrays, all cells start with the default value, here 0. The line `matrix[1][2] = 6` writes to the third column of the second row. Because indices again start at 0, “row 2” corresponds to index 1 and “column 3” to index 2.

The order of row and column is a convention, but it must be consistent. In Java the first index selects the outer array and the second selects the inner array. In matrices this means that the first index is the row; it also agrees with the mathematical notation a_{ij}. In images the `[y][x]` order is used for the same reason: first the row (height), then the column (width).

3.3.6.2 Arrays of arrays

Two-dimensional arrays can also be given initial values with an initializer list. This time each row is written inside its own braces:

```
int[][] matrix = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
System.out.println(matrix.length + " "
        + matrix[0].length + " " + matrix[1][2]);
```

```
3 3 6
```

To understand the output, you need to know how Java builds a two-dimensional array. In Java there is no real "table". A 2D array is actually an **array of arrays**. `matrix` is an array of three elements, and each element is a reference to an `int[]`, that is, to a row.

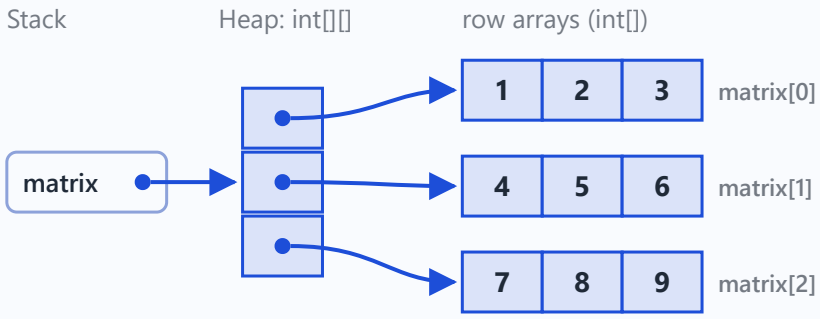


Figure 3.3.24: `matrix` is an array of row arrays. The three cells of the outer array point to three separate row arrays on the heap. `matrix[1]` is the second row itself.

Figure 3.3.24 shows this structure. `matrix.length` is the length of the outer array, that is, the number of rows: 3. `matrix[0]` is the first row itself and is an `int[]`; `matrix[0].length` is the number of columns of the first row: 3. `matrix[1][2]` first selects the second row and then reads the third element of that row: 6.

This structure has practical consequences. `matrix[1]` is a meaningful expression on its own and gives a row as a one-dimensional array. `Arrays.toString(matrix[1])` prints `[4, 5, 6]`. A row can be passed directly to a method that expects a one-dimensional array. On the other hand, `Arrays.toString(matrix)` prints not the contents of the rows but their identifiers: `[[I@…, [I@…, [I@…]`. To print nested arrays, `Arrays.deepToString` is used.

Example 3.3.9

Find the trace of the matrix `int[][] m = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};`, that is, the sum of its diagonal elements.

On the diagonal the row and column indices are equal: `m[0][0]`, `m[1][1]`, `m[2][2]`. So a single loop is enough: `for (int i = 0; i < m.length; i++) trace += m[i][i];`.

The passes make `trace` 1, 6 and 15. The trace of the matrix is 1 + 5 + 9 = 15. The trace is defined only for a square matrix; in a non-square matrix `m[i][i]` may go outside a row.

3.3.6.3 Jagged arrays and Pascal's triangle

Because the rows are separate arrays, their lengths do not have to be equal. A multidimensional array whose rows have different lengths is called a **jagged array**.

```
int[][] triangle = new int[3][];
for (int i = 0; i < triangle.length; i++) {
    triangle[i] = new int[i + 1];
}
System.out.println(triangle[0].length + " " + triangle[1].length
        + " " + triangle[2].length);
int[][] table = {{1, 2}, {3, 4, 5}};
for (int[] row : table) {
    for (int x : row) {
        System.out.print(x + " ");
    }
    System.out.println();
}
```

```
1 2 3
1 2
3 4 5
```

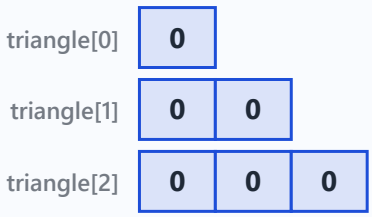


Figure 3.3.25: A jagged array: the row lengths of `triangle` are 1, 2 and 3.

`new int[3][]` gives only the first dimension. Three row references are created, but the rows themselves are not; all three references are `null`. The rows are allocated one by one in a loop: `triangle[i] = new int[i + 1]`. In the end the row lengths are 1, 2 and 3. If a row is accessed before it is allocated, a `NullPointerException` occurs. The second dimension alone cannot be given: `new int[][3]` does not compile.

The second part of the program builds a jagged array with an initializer list: the first row of `table` has two elements and the second has three. When traversing with `for-each`, the element of the outer loop is an `int[]` row; the inner loop goes over the elements of that row. In jagged arrays the bound of the inner loop must be separate for each row. When writing with a classic `for`, `matrix[i].length` is therefore used. Writing a fixed number of columns causes an out-of-bounds error on the short rows.

A nice application of a jagged array is Pascal's triangle. Each row of the triangle is one element longer than the previous one. The two ends of a row are 1; the inner elements are the sum of the two elements standing immediately to the left and immediately above in the row above.

```
int[][] p = new int[4][];
for (int i = 0; i < p.length; i++) {
    p[i] = new int[i + 1];
    p[i][0] = 1;
    p[i][i] = 1;
    for (int j = 1; j < i; j++) {
        p[i][j] = p[i - 1][j - 1] + p[i - 1][j];
    }
}
System.out.println(Arrays.toString(p[3]));
```

The program prints `[1, 3, 3, 1]`. This row holds the coefficients of the expansion of $(a + b)^3$.

`i`	**Creating the row**	**Ends**	**Inner cells (`j` = 1 … `i` − 1)**	`p[i]`
0	`new int[1]`	`p[0][0] = 1` (both ends are the same cell)	none	[1]
1	`new int[2]`	`p[1][0] = p[1][1] = 1`	none (`j < 1` is false at once)	[1, 1]
2	`new int[3]`	1, 1	`p[2][1]` = 1 + 1 = 2	[1, 2, 1]
3	`new int[4]`	1, 1	`p[3][1]` = 1 + 2 = 3, `p[3][2]` = 2 + 1 = 3	[1, 3, 3, 1]

Table 3.3.16: Building Pascal's triangle row by row.

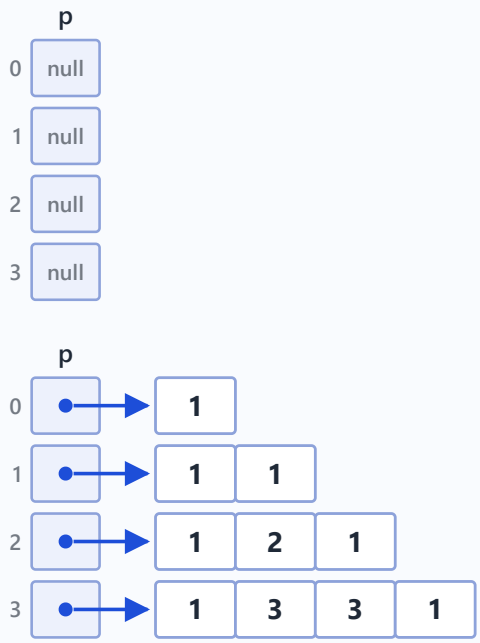


Figure 3.3.26: Pascal's triangle. (a) `new int[4][]` sets aside only four row references; all are `null`. (b) When the loop ends the row lengths are 1, 2, 3 and 4.

The subtle part of the code is the bounds of the inner loop. `j` starts at 1 and continues while it stays less than `i`; this way the end cells are not touched. In the rows `i` = 0 and `i` = 1 there are no inner cells and the inner loop never runs. The expression `p[i - 1][j]` is also safe: since `j` is at most `i - 1`, it does not pass the last index of the previous row. Row `i` has `i + 1` elements; the total number of elements is $1 + 2 + \ldots + n = n(n + 1)/2$. Using a jagged array instead of a square matrix saves roughly half the memory.

3.3.6.4 Filling and printing a matrix

The general way to traverse a two-dimensional array is two nested loops. The outer loop goes over the rows and the inner loop over the columns of a row.

```
public class MultiDimExample {
    public static void main(String[] args) {
        // a two-dimensional array (matrix) of size 3x3
        int[][] matrix = new int[3][3];
        // Assigning initial values to the elements of the matrix
        for (int row = 0; row < 3; row++) {
            for (int col = 0; col < 3; col++) {
                matrix[row][col] = row * 3 + col + 1;
            }
        }
        // Printing the matrix to the screen
        System.out.println("Matrix contents:");
        for (int row = 0; row < 3; row++) {
            for (int col = 0; col < 3; col++) {
                System.out.print(matrix[row][col] + " ");
            }
            System.out.println(); // Moving to a new line
        }
    }
}
```

```
Matrix contents:
1 2 3
4 5 6
7 8 9
```

In the filling loop `row * 3 + col + 1` is written to each cell. This expression numbers the cells row by row from 1 to 9. `row * 3` is the number of cells in the earlier rows; there are three cells in each row. `col` adds the position within the row. `+ 1` makes the numbering start from 1 rather than 0.

`row`	`col` = **0**	`col` = **1**	`col` = **2**
0	0 + 0 + 1 = 1	0 + 1 + 1 = 2	0 + 2 + 1 = 3
1	3 + 0 + 1 = 4	3 + 1 + 1 = 5	3 + 2 + 1 = 6
2	6 + 0 + 1 = 7	6 + 1 + 1 = 8	6 + 2 + 1 = 9

Table 3.3.17: Trace of the filling loop; `row * 3 + col + 1`.

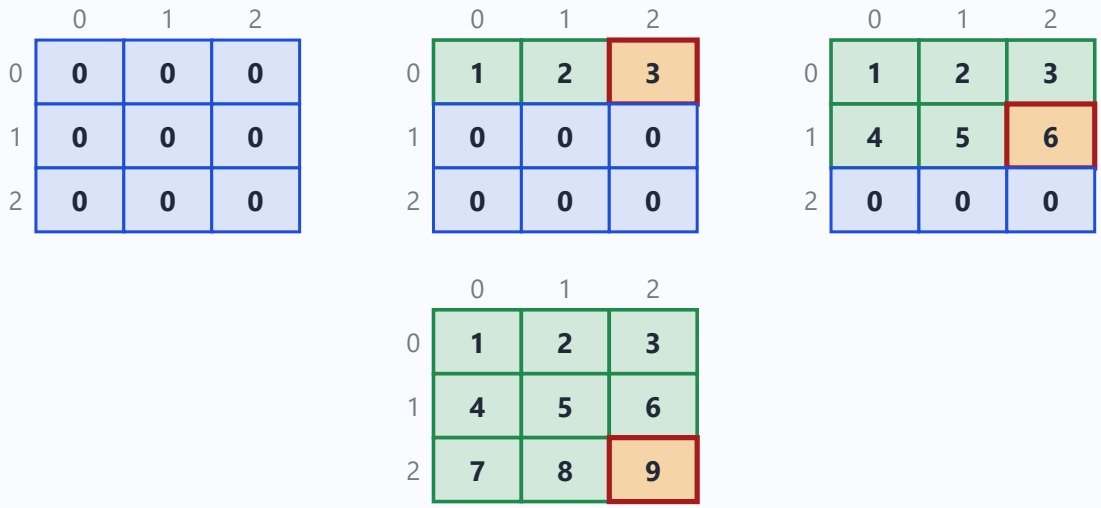


Figure 3.3.27: Filling a matrix. (a) `new int[3][3]` : all cells 0. (b) The first row is filled: 1, 2, 3. (c) The second row is filled. (d) The last cell `matrix[2][2]` = 9.

The cells are filled row by row: first `[0][0]` , `[0][1]` , `[0][2]` ; then the inner loop ends and the outer loop moves on to the next row. This order is called row-major traversal. The printing loop goes through in the same order. The elements of each row are written side by side with `print` ; when the inner loop ends, the `println()` in the outer loop moves to the next line. If `println()` were moved inside the inner loop, each number would be written on a separate line.

The cost of nested loops is the product of the number of rows and the number of columns. Traversing an $r \times c$ matrix takes $r \cdot c$ steps. For a square $n \times n$ matrix this is $O(n^2)$. The matrix itself holds n^2 elements; there is no cheaper way to touch each element once.

Example 3.3.10

Let the sum of each row of the matrix `int[][] m = {{1, 2, 3}, {4, 5, 6}};` be written into an array.

The result array needs as many cells as there are rows: `int[] t = new int[m.length];` . In the nested loop `t[i] += m[i][j];` is written. The bound of the inner loop is `m[i].length` .

The sum of the first row is 1 + 2 + 3 = 6 and of the second 4 + 5 + 6 = 15. The result is `t = [6, 15]` . The sum of two matrices is computed with the same method: `c[i][j] = a[i][j] + b[i][j]` . The sum of `{{1, 2}, {3, 4}}` and `{{5, 6}, {7, 8}}` is `[[6, 8], [10, 12]]` .

3.3.6.5 Three and more dimensions

Arrays of three or more dimensions can be used in Java as well. Each extra dimension means one more pair of square brackets. A three-dimensional array is an array of two-dimensional arrays.

```
// a three-dimensional array of size 3x3x3
int[][][] arr3d = new int[3][3][3];
arr3d[1][2][0] = 16; // layer 1, row 2, column 0
```

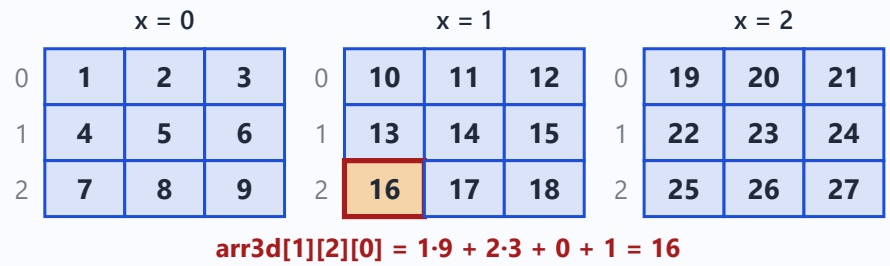


Figure 3.3.28: A 3×3×3 array: three layers, each a 3×3 matrix. `arr3d[1][2][0]` is the first cell of the third row of the second layer.

More dimensions allow more complex data. The rooms of a building can be kept as `[floor][row][col]`, a color image as `[y][x][channel]`, a table of measurements that change over time as `[time][row][col]`. The number of elements is the product of the dimensions: `new int[3][3][3]` holds 3 × 3 × 3 = 27 cells. As the number of dimensions grows, memory grows quickly. A color image of 1920 × 1080 kept as `int[1080][1920][3]` needs 6,220,800 cells, about 25 MB.

```
int[][][] arr3d = new int[3][3][3];
for (int x = 0; x < 3; x++) {
    for (int y = 0; y < 3; y++) {
        for (int z = 0; z < 3; z++) {
            arr3d[x][y][z] = x * 9 + y * 3 + z + 1;
        }
    }
}
System.out.println("3D array contents:");
for (int x = 0; x < 3; x++) {
    for (int y = 0; y < 3; y++) {
        for (int z = 0; z < 3; z++) {
            System.out.print(arr3d[x][y][z] + " ");
        }
        System.out.println();
    }
    System.out.println();
}
```

```
3D array contents:
1 2 3
4 5 6
7 8 9

10 11 12
13 14 15
16 17 18

19 20 21
22 23 24
25 26 27
```

Three nested loops traverse the array. The innermost `z` goes over the columns of a row, the middle `y` over the rows of a layer, and the outermost `x` over the layers. `x * 9 + y * 3 + z + 1` is written to each cell. 9 is the number of cells in a layer; the formula is the matrix-filling pattern `row * 3 + col + 1` extended by one dimension. As a result the cells are numbered from 1 to 27. `arr3d[1][2][0]` = 1 · 9 + 2 · 3 + 0 + 1 = 16. After each layer, the `println()` in the outer loop prints an empty line and the layers are separated from each other in the output.

3.3.6.6 Matrix transpose

Definition 3.3.14

The **transpose** *T* of a matrix *M* is the matrix in which rows have become columns and columns have become rows: *T[j][i] = M[i][j]*. The transpose of an *r* × *c* matrix is *c* × *r*.

```
public static void main(String[] args) {
    int[][] matrix = {
        {1, 2, 3},
        {4, 5, 6}
    };
    System.out.println("Original matrix:");
    printMatrix(matrix);
    int rows = matrix.length;
    int cols = matrix[0].length;
    int[][] transposed = new int[cols][rows];
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            transposed[j][i] = matrix[i][j];
        }
    }
    System.out.println("Transposed matrix:");
    printMatrix(transposed);
}
```

```
Original matrix:
1 2 3
4 5 6
Transposed matrix:
1 4
2 5
3 6
```

The program first reads the dimensions from the array: `rows = matrix.length` = 2, `cols = matrix[0].length` = 3. For the transpose a new array is needed, and its dimensions are in reverse order: `new int[cols][rows]`, that is, 3 × 2. A non-square matrix cannot be transposed in place, because the result has a different shape.

`i`	`j`	**Read**	**Written**
0	0	`matrix[0][0]` = 1	`transposed[0][0]`
0	1	`matrix[0][1]` = 2	`transposed[1][0]`
0	2	`matrix[0][2]` = 3	`transposed[2][0]`
1	0	`matrix[1][0]` = 4	`transposed[0][1]`
1	1	`matrix[1][1]` = 5	`transposed[1][1]`
1	2	`matrix[1][2]` = 6	`transposed[2][1]`

Table 3.3.18: Copying the transpose cell by cell.

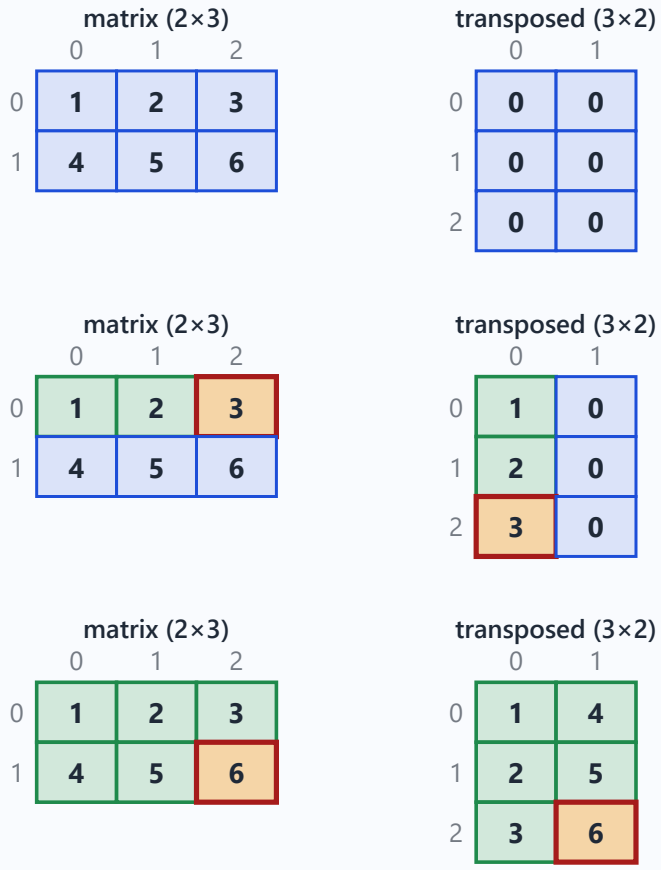


Figure 3.3.29: The transpose. (a) The 2×3 `matrix` and the 3×2 `transposed` filled with 0s. (b) The first row of `matrix` has become the first column of the transpose. (c) The last copy: `matrix[1][2]` = 6 is written to `transposed[2][1]`.

The loops go through the original matrix row by row. Each element is written to the cell of the transpose with its indices swapped. Table 3.3.18 shows this swap: when `i` = 0, the first row of `matrix` is read and written to the first column of `transposed`. The diagonal cell `[1][1]` stays in place. The amount of work is proportional to the number of elements: $r \cdot c$ copies.

Common mistake

Creating the transpose array with the same dimensions as the original matrix: `new int[rows][cols]`. For a 2×3 matrix this means a 2×3 array. When a write to `transposed[2][0]` is attempted, the row index 2 is out of bounds and an `ArrayIndexOutOfBoundsException` occurs. With square matrices this error is invisible; the code reveals it only when tried with a non-square matrix.

3.3.6.7 A helper method: `printMatrix`

The transpose program prints the matrix twice: first the original matrix and then the transpose. Instead of writing the printing code twice, the job has been moved into a helper method:

```
public static void printMatrix(int[][] matrix) {
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[i].length; j++) {
            System.out.print(matrix[i][j] + " ");
        }
        System.out.println();
    }
}
```

The parameter of the method is `int[][] matrix`, and it works with a two-dimensional array of any dimensions. The reason is that the number of rows is read from `matrix.length` and the number of columns from `matrix[i].length`, that is, from the array. The same method prints both the 2×3 original matrix and the 3×2 transpose correctly. Given a jagged array, it would print that correctly too.

What goes to the method is not a copy of the array but its reference. So the call is cheap even for a large matrix: only an address is copied. Because the method does not change the array, passing a copy of the reference has no side effect either.

What would happen if the bounds were written as constants? A printing method whose loops are written with `i < 3` and `j < 3`, when applied to the 3×2 transposed matrix, tries to access the third element of the first row:

```
0 0 Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 2 out of
bounds for length 2
        at TransposeFixed.print(TransposeFixed.java:9)
        at TransposeFixed.main(TransposeFixed.java:4)
```

After two 0s are printed, the exception occurs. The stack trace has two lines: the error is on line 9 of the `print` method, and this method was called from line 4 of `main`. A stack trace is read from the innermost call outward. Using `length` instead of a fixed number makes the method work correctly with any dimensions.

3.3.7 Application: Tic Tac Toe

This section combines what has been learned about arrays in a single program: a two-dimensional array, nested loops, passing an array to a method, input from the user and bounds checking. The program is the game of Tic Tac Toe (X-O), played by two people on the same computer.

3.3.7.1 The game and its data structure

The game is played on a 3×3 board. Two players, X and O, take turns putting their own mark in an empty cell. The one who places three of their marks in the same row, the same column or the same diagonal wins the game. If no empty cell remains and nobody has won, the game ends in a draw.

The most natural data structure for the board is `char[][] board = new char[3][3];`. Each cell holds one character: `'X'`, `'O'` or `' '` for an empty cell. A move is a single assignment: `board[row][col] = player`. The whole state of the game consists of the board and the information about whose turn it is.

3.3.7.2 `main` and splitting into methods

```
import java.util.Scanner;

public class TicTacToe {
    static Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {
        char[][] board = new char[3][3]; // 3x3 board
        char player = 'X'; // the first player is X
        boolean running = true;
        initBoard(board); // all cells empty
        while (running) {
            printBoard(board);
            makeMove(board, player);
            running = isRunning(board, player);
            // switch players
            player = (player == 'X') ? 'O' : 'X';
        }
        printBoard(board);
    }
    // ... methods
}
```

`main` is short and readable, because the job is split into four methods: filling the board, showing it, taking a move, and checking whether the game goes on. The loop repeats these steps as long as `running` is true. At the end of each turn play passes to the other player. `(player == 'X') ? 'O' : 'X'` is the conditional operator: it gives O if the player is X, and X otherwise.

A copy of the reference to the board goes to the methods. When `makeMove` writes to a cell of the board, the change is visible in the board in `main`, because there is only one array. This is the useful side of arrays being reference types: methods can change shared state.

The `Scanner` object is declared as a `static` field at the top of the class. This way all the methods use the same `Scanner`. The methods themselves are `static` too. Since `main` is static, the methods it calls without creating an object must be static as well. If the word `static` is forgotten, the compiler

gives this message:

```
E_Static.java:4: error: non-static method initBoard(char[][]) cannot be referenced from a
static context
        initBoard(board);
        ^
1 error
```

3.3.7.3 Filling and showing the board

```
public static void initBoard(char[][] board) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            board[i][j] = ' ';
        }
    }
}

public static void printBoard(char[][] board) {
    System.out.println("-------------");
    for (int i = 0; i < 3; i++) {
        System.out.print("| ");
        for (int j = 0; j < 3; j++) {
            System.out.print(board[i][j] + " | ");
        }
        System.out.println("\n-------------");
    }
}
```

The empty board looks like this:

```
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
```

The cells of a board created with `new char[3][3]` start with `'\u0000'`. This character is invisible on the screen, or appears as a strange symbol in some terminals. Also, the check for an empty cell will compare against `' '`. That is why `initBoard` first writes the space character to all the cells.

`printBoard` prints each row in the form `| X | O | |`. At the start of the row `"| "` is printed, and after each cell `" | "`. The string `"\n-------------"` at the end of the row does two jobs with a single `println`: `\n` ends the row, then the separator line follows, and `println` moves to the next line.

3.3.7.4 Taking a move

```
static Scanner scanner = new Scanner(System.in);

public static void makeMove(char[][] board, char player) {
    int row, col;
    do {
        System.out.print("Player " + player
                + " 's turn. Enter row and column (1-3): ");
        row = scanner.nextInt() - 1;
        col = scanner.nextInt() - 1;
    } while (row < 0 || row > 2 || col < 0 || col > 2
            || board[row][col] != ' ');
    board[row][col] = player;
}
```

The user enters the row and column between 1 and 3, because people count from 1. The code turns these values into indices 0–2 with `- 1`. The input is taken with a `do-while` loop. `do-while` runs the body at least once and checks the condition at the end. The question is asked first; if an invalid or occupied cell was chosen, the condition is true and the question is asked again.

The order in the condition is deliberate. The four bounds checks come first, and the cell check is written last. The `||` operator does **short-circuit** evaluation: it goes from left to right and stops at the first true condition. If the user enters 4 5, `row` becomes 3 and `col` becomes 4. `row > 2` is true and evaluation stops there. `board[3][4]` is never read, no exception occurs, and the question is asked again. If the cell check had been written before the bounds checks, the program would have crashed with an `ArrayIndexOutOfBoundsException`.

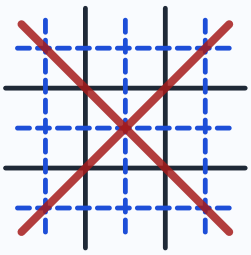
Common mistake

Writing the condition in the order `board[row][col] != ' ' || row < 0 || …`. Short-circuiting works only from left to right; since the access to the cell is at the far left, it is made before the bounds checks. With valid input the program runs without trouble, and the bug shows up only when the user enters a number outside the range. That is why boundary values (0, 4, −1) must be tried separately in testing.

3.3.7.5 Winner and draw

After each move the game answers two questions: is there a winner, and is there an empty cell left? The `isRunning` method first checks for a win:

```
public static boolean isRunning(char[][] board, char player) {
    // Check the winning cases
    if ((board[0][0] == player && board[0][1] == player && board[0][2] == player) ||
        (board[1][0] == player && board[1][1] == player && board[1][2] == player) ||
        (board[2][0] == player && board[2][1] == player && board[2][2] == player) ||
        (board[0][0] == player && board[1][0] == player && board[2][0] == player) ||
        (board[0][1] == player && board[1][1] == player && board[2][1] == player) ||
        (board[0][2] == player && board[1][2] == player && board[2][2] == player) ||
        (board[0][0] == player && board[1][1] == player && board[2][2] == player) ||
        (board[0][2] == player && board[1][1] == player && board[2][0] == player)) {
        System.out.println("Congratulations, player " + player + " wins!");
        return false;
    }
    // ... draw check
```



3 rows + 3 columns + 2 diagonals

Figure 3.3.30: The eight winning lines: 3 rows, 3 columns and 2 diagonals.

The eight winning lines are checked one by one. Each line requires all three cells to hold the player's mark; these conditions are joined with `&&`. One line is enough; the lines are joined with `||`. Only the player who made the last move is checked. Only that player can have completed a new line; if the other player had won, the game would have ended on the previous move.

If there is a winner, the method returns `false`. This looks backwards at first sight, but the name of the method is the question “is the game still going?” and the answer is “no”. The long conditional expression could also have been written with loops: one loop for the three rows and three columns, and a separate check for the two diagonals. The explicit form is long, but it shows directly which lines are checked.

If there is no winner, a draw is checked for:

```
// the rest of the isRunning method
// Check for a draw
boolean draw = true;
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if (board[i][j] == ' ') {
            draw = false;
            break;
        }
    }
    if (!draw) {
        break;
    }
}
if (draw) {
    System.out.println("The game is a draw.");
    return false;
}
return true;
```

This code starts with an assumption: the game has ended in a draw (`draw = true`). A single empty cell refutes this assumption. When the inner loop finds an empty cell, it sets `draw` to `false` and leaves with `break` . `break` leaves only the loop it is in. The outer loop keeps running. That is why a second check is needed in the body of the outer loop: `if (!draw) break;` .

The order of the checks matters too. A win is checked first, then a draw. The last move may fill the board and complete a line at the same time. If the order were reversed, such a game would wrongly be declared a draw.

To see how the two `break` s work, let us trace only the draw code with a board that has a single empty cell:

```
boolean draw = true;
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if (board[i][j] == ' ') {
            draw = false;
            break;
        }
    }
    if (!draw) {
        break;
    }
}
System.out.println(draw ? "draw" : "continue");
```

`i`	`j`	`board[i][j]`	**Operation**	`draw`
0	0	`'X'`	not empty	true
0	1	`'O'`	not empty	true
0	2	`'X'`	the inner loop ends by itself; `!draw` is false	true
1	0	`'O'`	not empty	true
1	1	`'X'`	not empty	true
1	2	`' '`	empty cell: `draw = false`, inner `break`	false
1	—	—	`!draw` is true: outer `break`	false

Table 3.3.19: Trace of the draw check; board `X O X` · `O X ␣` · `O X O` (␣: space).

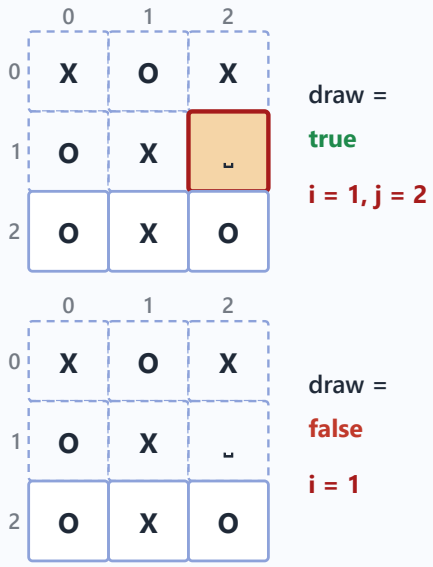


Figure 3.3.31: The draw check. (a) A space was found at cell `(1, 2)`. (b) Outer `break`: row 2 is never checked.

The program prints `continue`. Of the nine cells only six were looked at; row 2 is never checked. Once an empty cell has been found, continuing the search is pointless, because the result can no longer change. This kind of “stop at the first counterexample” pattern is the two-dimensional counterpart of the `break` in linear search.

3.3.7.6 A sample game

Let a game be played with the following inputs. Each pair is the row and column choice of the player whose turn it is:

```
1 1   1 2   2 2   2 2   1 3   3 3
```

Player	Input	Assignment	Result
X	1 1	`board[0][0] = 'X'`	no winner, empty cells remain; play passes to O
O	1 2	`board[0][1] = 'O'`	play passes to X
X	2 2	`board[1][1] = 'X'`	play passes to O
O	2 2	—	`board[1][1]` is occupied: the question is asked again
O	1 3	`board[0][2] = 'O'`	play passes to X
X	3 3	`board[2][2] = 'X'`	main diagonal is X: the winning message, the loop ends

Table 3.3.20: The moves of the sample game.

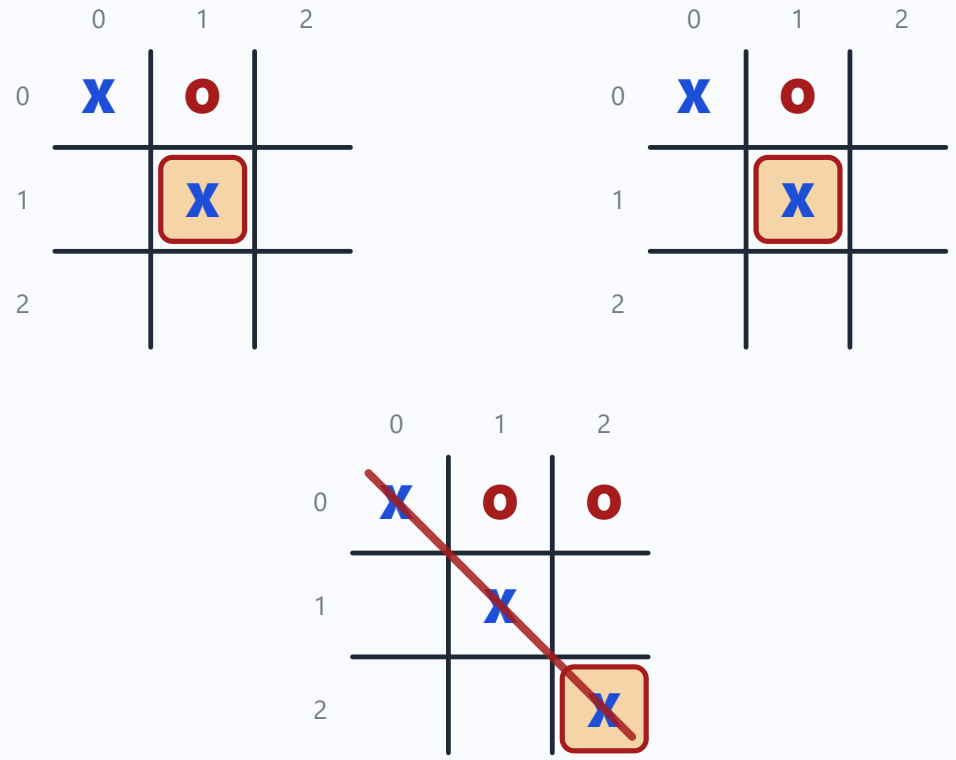


Figure 3.3.32: The sample game. (a) The board after three moves. (b) O chooses the occupied cell (2, 2); the `do-while` condition is true and the question is asked again. (c) X completes the main diagonal.

The final output of the program:

```
Congratulations, player X wins!
-------------
| X | O | O |
-------------
|   | X |   |
-------------
|   |   | X |
-------------
```

In the fourth move the cell chosen by O is occupied. `board[1][1] != ' '` is true and `do-while` asks the same question once more. The turn does not change in the meantime, because `makeMove` does not return until a valid move has been made. In the last move X completes the main diagonal (`[0][0]` , `[1][1]` , `[2][2]`). `isRunning` prints the winning message and returns `false` . The loop in `main` ends and the board is shown for the last time.

The program has aspects that could be improved. If the user types a letter instead of a number, `nextInt()` throws `InputMismatchException` and the program stops. The draw check could be moved to a separate `isBoardFull` method; then `isRunning` would be shorter and each method would answer a single question. The win check could also be narrowed to look at the row, the column and, if necessary, the diagonals of the last move.

3.3.8 Common mistakes

Array mistakes fall into two groups. Compile errors show up before the program runs, with the messages given by javac, and are relatively easy to fix. Run-time errors appear while the program runs, as an exception, or produce a wrong result without any message at all. The following two tables bring together the mistakes that were treated separately in the body of the notes. The messages were obtained with the javac and java tools of JDK 21.

Mistake	Code	javac message	Fix
Giving the short initializer later	`a = {1, 2, 3};`	`illegal start of expression`	`a = new int[]{1, 2, 3};`
Both size and list	`new int[3]{1, 2, 3}`	`array creation with both dimension expression and initialization is illegal`	Give only one of them.
Forgetting the size	`new int[]`	`array dimension missing`	`new int[n]`
Element of the wrong type	`int[] a = {1, 2.5, 3};`	`incompatible types: possible lossy conversion from double to int`	`double[]` or integer values
Writing `length()`	`a.length()`	`cannot find symbol … method length()`	`a.length`
C-style size	`int a[3] = {…};`	`']' expected`	`int[] a = {…};`
Variable declared inside a branch	`if (…) { double median = …; }`	`cannot find symbol … variable median`	Declare the variable outside the branches.
Method that is not `static`	`public void initBoard(…)`	`non-static method … cannot be referenced from a static context`	`public static void …`

Table 3.3.21: Common compile errors with arrays.

Mistake	Result	Fix
Thinking the last index is `length` : `a[a.length]` , `i <= a.length`	`ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3`	`a[a.length - 1]` , `i < a.length`
An array that was never created: `int[] a = null; a[0]`	`NullPointerException: Cannot load from int array …`	The array is first created with `new` .

Mistake	Result	Fix
Printing an array directly: `println(a)`	An identifier such as `[I@1c20c684`	`Arrays.toString(a)`
Mistaking assignment for copying: `b = a`	Changing `b[0]` changes `a[0]` too.	`a.clone()`, `Arrays.copyOf`
Comparing contents with `==`	`false` for two arrays with the same elements	`Arrays.equals(a, b)`
Thinking the first index is 1	The first element is skipped or the bound is exceeded.	`arr[0]` in the code; `(i + 1) + ". element"` on the screen
Writing to an array with `for-each`	The array does not change; only the copy does.	Classic `for` and `arr[i] = …`
Binary search on an unsorted array	Meaningless result, no warning	`Arrays.sort` first
Mean with integer division	29 / 6 = 4, then 4.0	`sum / (double) n`

Table 3.3.22: Common run-time and logic errors with arrays.

Most of these mistakes come down to two sources. The first is that the index starts at 0 and the last index is `length - 1`. The second is that an array variable holds a reference, not a value. If these two points are checked while array code is written, most mistakes are prevented as it is written. When the code is tried out, the boundary cases must also be tested separately: the empty array, the array with one element, the cases where the target is the first and the last element, odd and even lengths.

3.3.9 Summary

- An array is a structure that holds a fixed number of elements of the same type in consecutive cells. Elements are accessed by an index starting at 0; the last index of an array with *n* elements is *n − 1*.
- An array is created with `new int[n]` and its cells get default values. This value is 0 or 0.0 for numbers, `false` for `boolean`, `'\u0000'` for `char`, and `null` for objects.
- The initializer list `{1, 2, 3}` is used only on the declaration line; elsewhere `new int[]{1, 2, 3}` is written. The size and the list cannot be given together.
- `length` is a property and is written without parentheses. All the indices of an array are traversed with the pattern `for (int i = 0; i < arr.length; i++)`.
- The `for-each` loop gives a copy of the next element on each pass. It is suitable for reading; for writing to the array and for jobs that need the index, the classic `for` is used.
- An array variable is a reference. `b = a` creates a second name for the same array; `clone()` and `Arrays.copyOf` create a new array. `==` compares addresses, `Arrays.equals` compares contents.
- When an array is passed to a method, a copy of the reference goes. If the method changes the cells, the change is visible in the caller; assigning a new array to the parameter does not affect the caller.
- All arrays are on the heap; only references sit on the stack. Unreachable arrays are cleaned up by the garbage collector.

- An out-of-bounds index causes `ArrayIndexOutOfBoundsException`. The message gives the faulty index, the length and the line number.
- The mean is found in one pass in O(n), the median with the middle element after sorting, and the mode with the longest run in the sorted array. The median and the mode are O(n log n) because of the sorting.
- Access is O(1), binary search O(log n), linear search and reversal O(n), and matching, selection sort and bubble sort O(n^2). Binary search works only on a sorted array.
- A multidimensional array is an array of arrays. `m.length` is the number of rows and `m[i].length` is the number of columns of that row. Rows can have different lengths; that is why `m[i].length` is used in loops.
- Tic Tac Toe combines many techniques: a `char[][]` board, state shared by reference, input validation with `do-while`, bounds checking with short-circuiting, and leaving nested loops with two `break`s.

3.3.10 Key terms

array element index array initializer default value for-each loop reference type null reference pass by value stack (memory) heap (memory) garbage collector dynamic array linear search binary search time complexity (Big-O) selection sort bubble sort two-dimensional array jagged array short-circuit evaluation

---

3.3.11 Test Yourself

1. With `int[] d = {3, 8, 1, 9, 4};`, what is `s` after the loop `for (int i = d.length - 1; i >= 0; i -= 2) s += d[i];`? (`s` starts at 0.)

   Answer

   The loop starts from the last index, 4, and decreases by two: `i` = 4, 2, 0. The elements read are `d[4]` = 4, `d[2]` = 1 and `d[0]` = 3. When `i` becomes −2 the condition is false. `s` = 4 + 1 + 3 = **8**. Had `d.length` been written as the start, `d[5]` would be accessed on the first pass and the exception would occur.

2. What is printed after the following lines? `int[] x = {2, 4, 6}; int[] y = x; int[] z = x.clone(); y[1] = 0; z[2] = 9;` followed by `Arrays.toString(x)`, `x == y` and

`Arrays.equals(x, z)`.

Answer

`y` points to the same array as `x`; `y[1] = 0` changes this array and `x` = `[2, 0, 6]`. `z` is a separate copy; `z[2] = 9` changes only the copy and `z` = `[2, 4, 9]`. `x == y` is **true**. `Arrays.equals(x, z)` is **false**, because the contents now differ.

3. 23 is searched for with binary search in the sorted array `{2, 5, 8, 12, 16, 23, 38, 56, 72, 91}`. Which middle indices are looked at? How many comparisons are made?

Answer

`lo` = 0, `hi` = 9 → `mid` = 4 (16); 23 > 16, `lo` = 5. `mid` = (5 + 9) / 2 = 7 (56); 23 < 56, `hi` = 6. `mid` = (5 + 6) / 2 = 5 (23): found. The middle indices are 4, 7 and 5. **3 comparisons** are made and the method returns 5. If 40 were searched for, `mid` = 6 (38) in the fourth step, then the space would be empty and −1 would be returned.

4. The array `{5, 1, 4, 2, 8}` is sorted with selection sort. What does the array look like at the end of each pass? How many comparisons are made?

Answer

`i` = 0: the smallest is 1 (index 1), swap: `[1, 5, 4, 2, 8]`. `i` = 1: the smallest is 2 (index 3), swap: `[1, 2, 4, 5, 8]`. `i` = 2: the smallest, 4, is already in place. `i` = 3: the smallest, 5, is already in place. The array was sorted after the second pass, but selection sort does not notice. The number of comparisons is 4 + 3 + 2 + 1 = **10** in every case.

5. For `int[][] m = {{1, 2}, {3, 4}, {5, 6}};`, what are `m.length`, `m[0].length` and the sum of the second column?

Answer

`m` is an array with three rows: `m.length` = **3**. The first row has two elements: `m[0].length` = **2**. The index of the second column is 1: `m[0][1] + m[1][1] + m[2][1]` = 2 + 4 + 6 = **12**. The loop is written `for (int i = 0; i < m.length; i++) t += m[i][1];`; the row changes and the column is fixed.

6. After the lines `int[][] r = new int[3][]; r[0] = new int[]{1}; r[2] = new int[2];`, what

does `r[2].length + r[1].length` give?

Answer

It gives no number; the program stops with a `NullPointerException`. `new int[3][]` creates only three row references. Rows have been assigned to `r[0]` and `r[2]`, but `r[1]` is still `null`. `r[1].length` tries to read a field of a `null` reference. The message has the form `Cannot read the array length because "…[1]" is null`.

7. Let `static void f(int[] d) { d = new int[]{9, 9}; }` and `static void g(int[] d) { d[0] = 9; }` be given. With `int[] a = {1, 2};`, first `f(a)` and then `g(a)` are called. What is `a` after each call?

Answer

After `f(a)`, `a` is still `[1, 2]`. The `d` in `f` is a copy of the reference; assigning a new array binds only this copy to the new array. After `g(a)`, `a = [9, 2]`. The `d` in `g` points to the same array and `d[0] = 9` writes to a cell of that array.

8. The mode algorithm given in the notes is applied to the sorted array `{1, 1, 3, 3, 3, 5, 5, 5}`. What is the result? What would change if `>=` were written instead of `>`?

Answer

When the run of 3s reaches length 3, 3 > 2 is true and the longest run becomes 3 × 3. The run of 5s also reaches 3, but 3 > 3 is false. The result is **mode 3, count 3**; in a tie the first one found is kept. With `>=` the run of 5s would overtake the longest run at equal length and the result would be 5. This set actually has two modes.

9. In how many passes does bubble sort with the `swapped` flag finish on the array `{2, 1, 3, 4, 5}`? How many comparisons would be made without the flag?

Answer

In the first pass 2 and 1 are swapped (4 comparisons). In the second pass no swap is made (3 comparisons) and `break` runs: **2 passes, 7 comparisons**. Without the flag all the passes would run: 4 + 3 + 2 + 1 = **10** comparisons.

3.3.12 Further reading

- Oracle, *The Java Tutorials*, the pages “Arrays” and “The for Statement”: explain declaring arrays, the initializer list and multidimensional arrays. `java.util.Arrays` and the enhanced `for` are also covered with short examples. It is the official, free starting resource for Java.

- A. B. Downey and C. Mayfield, *Think Java* (2nd ed.), Chapter 7 “Arrays and References”: explains arrays through the concept of a reference. It complements these notes on aliasing, copying and passing an array to a method.
- P. Deitel and H. Deitel, *Java: How to Program* (11th ed.), Chapter 7 “Arrays and ArrayLists”: a long account with plenty of examples, which also covers the move from arrays to `ArrayList`.
- J. Gosling et al., *The Java Language Specification, Java SE 21 Edition*, Chapter 10 “Arrays”: the precise definition of array types, creation, default values and bounds checking. On a point of detail, the last word is in this document.

3.3.13 References

1. Oracle, *The Java Tutorials*: “Arrays”; “The for Statement” (enhanced for).
2. Oracle, *Java SE 21 API Specification*: `java.util.Arrays`.
3. J. Gosling et al., *The Java Language Specification, Java SE 21 Edition*, Oracle, 2023 (Chapter 10: Arrays).
4. P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson (Chapter 7: Arrays and ArrayLists).
5. K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly, 2022.
6. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson, 2022 (3.10 Arrays).
7. A. B. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019 (Chapter 7: Arrays and References).

PART I · CHAPTER 3.4

Strings

Most programs work with text. A user name, an e-mail address, a file path or a message written to the screen is always a piece of text made of characters. In Java such texts are stored in objects of the `String` class. These notes explain how strings are created, in what order concatenation with `+` proceeds, and how characters are accessed by index. The most frequently used string methods follow. The immutability of strings, the effect of the locale on Turkish letters and `StringBuilder` for concatenation in loops are treated in separate sections. After the finer points of reading text from the keyboard with `Scanner`, the comparison of strings is covered: the difference between `==` and `equals()`, the string pool and `compareTo`. The last sections cover formatted output with `printf`, the palindrome and word-frequency applications, and common mistakes. All the code in the slides was run with JDK 21; the outputs and error messages are taken from the real runs.

LEARNING OUTCOMES

Creating strings
Create `String` objects with string literals and with `new`; explain how `+` concatenation is evaluated from left to right.

Index and methods
Reach the parts of a string with `charAt`, `length`, `indexOf` and `substring`; traverse a string character by character with a loop.

Immutability
Show with memory diagrams that string methods return new objects; know the effect of the locale on case conversion; use `StringBuilder` in loops.

User input
Explain the difference between `next()`, `nextLine()` and `nextInt()`; avoid the newline trap.

Comparison
Explain the difference between `==` and `equals()` with the string pool; use `compareTo` and the other comparison methods correctly.

Formatted output
Adjust width, alignment and precision with `printf` and `String.format`.

3.4.1 The string concept

A computer stores all data as numbers. A letter, too, is a number in memory; the `char` type holds the code of a single character. Real programs, however, work not with single letters but with words and sentences. The name of a student, the description of a product or a reply from a server is a piece of text made of consecutive characters. Java offers a ready-made class for such texts. This section introduces that class and explains how a string is created and how its characters are accessed.

3.4.1.1 What is a string?

Definition 3.4.1

A **string** is an object that stores a piece of text made of characters. In Java the type of strings is the class `java.lang.String`. Text written between double quotes in the source code, such as `"Tarkan"`, is called a **string literal**.

A few details in the definition are easily overlooked. First, `String` is a **class**, not a primitive type. That is why its name starts with an uppercase letter: `int`, `double` and `char` are written in lowercase, while `String` has a capital S. A `String` variable stores not the text itself but the location in memory of the object that holds the text, that is, a **reference**. Second, the `String` class is in the `java.lang` package. This package is added to every program automatically, so no `import` line is needed. Third, `new` is not used to write a string literal. The line `String name = "Tarkan";` creates the object by itself.

Java stores characters in **Unicode**. Unicode is the standard that gives a number to the characters of the world's writing systems. Thanks to this, Turkish letters such as `ç`, `ğ`, `ı`, `İ`, `ö`, `ş` and `ü` sit in a string without trouble. A string actually consists of ordered character boxes. Each box has an ordinal number, and the numbering starts at 0 (Figure 3.4.1).



Figure 3.4.1: The string `"Tarkan"` consists of six character boxes. The numbers 0–5 below the boxes are the indices.

A single character and a one-character string are different things. `'T'` is written with single quotes and its type is `char`; in memory it is a two-byte value holding the number 84. `"T"` is written with double quotes and is a `String` object of length 1. The two cannot be used in place of each other. The line `char c = "T";` does not compile, and neither does `String s = 'T';`.

The first example in the slides creates three strings and prints them:

```
String name = "Tarkan";
String fullName = "Mustafa Sandal";
String point = "(" + 3 + ", " + 5 + ")";
System.out.println(name);
System.out.println(fullName);
System.out.println(point);
```

```
Tarkan
Mustafa Sandal
(3, 5)
```

In the first two lines string literals are assigned to variables. A space is also a character: `"Mustafa Sandal"` is a string of 14 characters. The third line builds a string from numbers. Because `"("` is a string, the `3` that follows is converted to text and appended; the same rule works for `", "`, `5` and `")"`. In the end the variable `point` refers to the text `"(3, 5)"`. This concatenation rule is treated in detail just below, because it is where beginners are most often surprised.

3.4.1.2 Creating strings

A string can be created in several ways. The most common is the string literal. Besides it, the constructors of the `String` class, the `String.valueOf` method, which turns a number into a string, and the `concat` method, which joins two strings end to end, are also used. The first half of the code in the slides shows these ways together:

```
String s0 = "Giresun";            // string literal
char[] letters = {'J', 'a', 'v', 'a'};
String s1 = new String(letters);  // char[]
String s2 = String.valueOf(42);   // from a number
String s3 = "Giresun".concat(" Üni.");
System.out.println(s1 + " " + s2 + " " + s3);

System.out.println("Total: " + 3 + 5);
System.out.println("Total: " + (3 + 5));
System.out.println(3 + 5 + " apples");
System.out.println('a' + 'b' + "c");
System.out.println("" + 'a' + 'b');
```

```
Java 42 Giresun Üni.
Total: 35
Total: 8
8 apples
195c
ab
```

The first six lines of the code work as follows:

- `String s0 = "Giresun";` is the most common form: the string literal is assigned directly to the variable.
- `new String(letters)` builds a new string from the characters in a `char` array. Because the array is `{'J', 'a', 'v', 'a'}`, `s1` becomes `"Java"`. This constructor is useful for turning a text whose characters are computed one by one into a string.
- `String.valueOf(42)` returns the text form of the number 42, that is, the string `"42"`. `valueOf` is a **class method**: it is called not through an object but with the class name.
- `"Giresun".concat(" Üni.")` joins two strings and returns `"Giresun Üni."`. Here the method is called directly on a string literal; this is valid because a literal is an object too.

The printing line joins these three strings with spaces between them: `Java 42 Giresun Üni.`. The second half of the code shows the behavior of the `+` operator. The `Total: 35` on the second line of the output surprises most students; the reason is the subject of the next subsection.

The form `new String("Giresun")` also compiles and runs, but it produces an unnecessary object. This form will be used in the comparison section to explain the string pool. In everyday code there is no benefit in writing `new` to create a string from a literal.

3.4.1.3 Concatenation and the order of the + operator

Definition 3.4.2

String concatenation means producing a new string by joining two values end to end. In Java, if at least one of the operands of the `+` operator is a `String`, the other operand is also converted to a string and the two texts are joined. If both operands are numbers, `+` performs numeric addition.

The same `+` sign does two different jobs: addition and concatenation. Which one is done is decided separately for each operation. Only the types of the two operands at that moment are looked at. Also, `+` is evaluated from left to right. That is, the expression `a + b + c` is computed as `(a + b) + c`. Considered together, these two rules explain all the outputs in the slides:

- `"Total: " + 3 + 5`: first `"Total: " + 3` is computed. Because the left side is a string, 3 is converted to text: `"Total: 3"`. Then comes `"Total: 3" + 5`, and concatenation is done again. The result is `Total: 35`.
- `"Total: " + (3 + 5)`: the parentheses are computed first. Since both operands inside are `int`, addition is performed and 8 comes out. Then 8 is appended to the string: `Total: 8`.
- `3 + 5 + " apples"`: the two operands on the left are numbers and are added first, giving 8. Only after that is a string met: `8 apples`.
- `'a' + 'b' + "c"`: `char` is a numeric type. When two `char`s are added, their codes are added: 97 + 98 = 195. Then `"c"` is appended: `195c`.
- `"" + 'a' + 'b'`: the empty string at the start changes everything. `"" + 'a'` is already a string, `"a"`. Then `'b'` is appended as text: `ab`.

In short the rule is this: the expression is read from left to right. Until a string is met, numbers are added. Every `+` after the moment a string appears is concatenation. If numbers are to be added, parentheses are used.

The trace slide in the slides shows the same rule with variables:

```
int a = 4, b = 2;
String s1 = "Result: " + a + b;
String s2 = "Result: " + (a + b);
String s3 = a + b + " pens";
String s4 = 'x' + 'y' + "z";
String s5 = "" + 'x' + 'y';
System.out.println(s1 + " | " + s2);
System.out.println(s3 + " | " + s4 + " | " + s5);
```

```
Result: 42 | Result: 6
6 pens | 241z | xy
```

Each row of Table 3.4.1 shows one `+` operation. The columns give the types of the operands and the result of that step.

Variable	Operation	Left operand	Right operand	Result
`s1`	`"Result: " + a`	String	int	concatenation: `"Result: 4"`
	`… + b`	String	int	concatenation: `"Result: 42"`
`s2`	`(a + b)`	int	int	addition: 6
	`"Result: " + 6`	String	int	concatenation: `"Result: 6"`
`s3`	`a + b`	int	int	addition: 6
	`6 + " pens"`	int	String	concatenation: `"6 pens"`

Variable	Operation	Left operand	Right operand	Result
`s4`	`'x' + 'y'`	char	char	addition: 120 + 121 = 241
	`241 + "z"`	int	String	concatenation: `"241z"`
`s5`	`"" + 'x'`	String	char	concatenation: `"x"`
	`"x" + 'y'`	String	char	concatenation: `"xy"`

Table 3.4.1: The kind and result of each `+` operation when `a = 4` and `b = 2`. The operations are performed from left to right.

The difference between `s1` and `s2` rests on a single pair of parentheses, but the results differ as much as 42 and 6. Figure 3.4.2 shows the first steps of these two lines with the drawings from the slides. Under the upper boxes are the types of the operands, and in the lower box are the result of that step and its type.

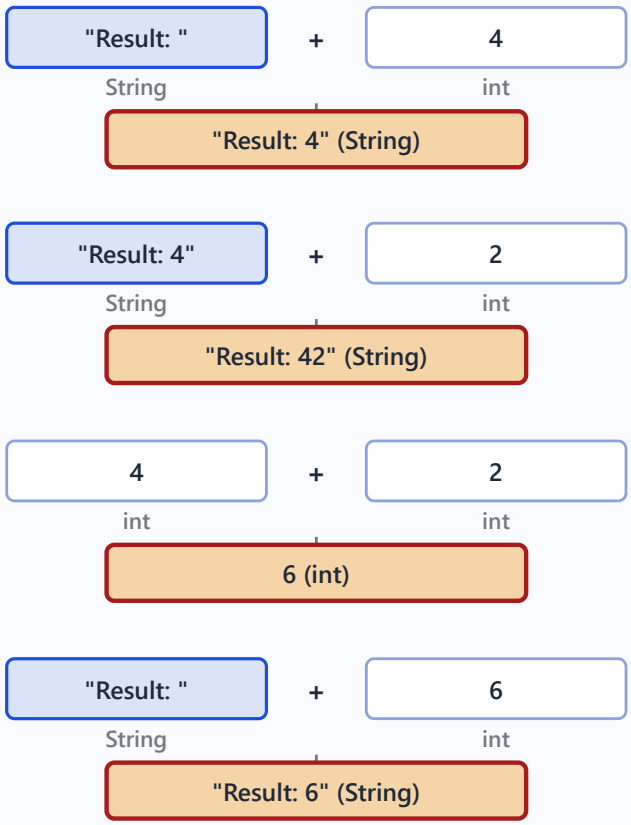


Figure 3.4.2: The steps of lines `s1` and `s2`. (a) Because the left side is a `String`, 4 is appended to the text. (b) The result is still a `String`; 2 is appended as text too and `"Result: 42"` comes out. (c) Inside the parentheses two `int`s are added and the result is 6. (d) 6 is appended to the string.

The line `s4` needs separate attention. Although `char` values look like text, Java handles them as numbers. The Unicode code of the character `'x'` is 120 and that of `'y'` is 121. When two `char`s are added side by side the result is an `int`, 241. This number is converted to text only when `"z"` is

met. In `s5`, on the other hand, the `""` at the start turns the first operation into concatenation, and the two letters come side by side as text. Putting an empty string in front is a common way of joining characters. The same job is also done with `String.valueOf('x') + 'y'`.

Common mistake

Forgetting the parentheses when printing a sum: the line `System.out.println("Total: " + x + y);` does not add `x` and `y`, it writes them side by side. With `x = 10`, `y = 5` the output is `Total: 105`. The correct form is `"Total: " + (x + y)`. The compiler misses the error, because both forms are valid Java.

One more detail: concatenation produces a **new** `String` object every time. In the expression `"Result: " + a + b` two intermediate strings are logically created. For a small number of pieces this does not matter. When concatenation is done thousands of times in a loop, the cost grows quickly. This cost is computed in the immutability section.

3.4.1.4 Indexing

Definition 3.4.3

The **index** is the ordinal number of a character in the string. The index of the first character is 0 and that of the last character is `length() - 1`. Accessing a character by its index is called **indexing**.

Indexing has two basic methods. `length()` returns the number of characters in the string. `charAt(i)` gives the character at index `i` as a `char`. `"Tarkan"` has six characters; its indices run from 0 to 5 (Figure 3.4.3). Although the length is 6, there is no character number 6. Because numbering starts at 0, the last index is always one less than the length.



Figure 3.4.3: The first character is obtained with `charAt(0)` and the last with `charAt(length() - 1)`, that is, `charAt(5)`.

The example in the slides tries these methods on `"Tarkan"`:

```
String name = "Tarkan";
System.out.println(name.length());
System.out.println(name.charAt(0));
System.out.println(name.charAt(name.length() - 1));
char c = name.charAt(2);
System.out.println(c + 1);
System.out.println((char) (c + 1));
```

```
6
T
n
115
s
```

The first line prints the length, 6. `charAt(0)` gives the first letter, `T`. For the last letter the index is not written by hand but computed from the length: `name.length() - 1` = 5 and `charAt(5)` = `n`. This form works for strings of any length. On the next line `c` takes the character `'r'` at index 2.

The last two lines show that `char` is a number. The Unicode code of the character `'r'` is 114. In the expression `c + 1` a `char` and an `int` are added and the result is an `int`: 115. A number, not a letter, appears on the screen. To see the next letter, the result is converted back with `(char)`: the character with code 115 is `s`. Simple encryption examples based on shifting letters use this conversion.

`length()` is a **method**, that is, it has parentheses at the end. For arrays, on the other hand, the length is a **field** and is written without parentheses: `arr.length`. The two forms are often confused and both give a compile error:

```
StringLength.java:4: error: cannot find symbol
        int n = s.length;
                ^
  symbol:   variable length
  location: variable s of type String
```

```
ArrayLength.java:4: error: cannot find symbol
        int n = d.length();
                ^
  symbol:   method length()
  location: variable d of type int[]
```

The `symbol` line in the messages tells the difference clearly. In the first, the compiler looks for a *variable* named `length` in `String` and cannot find it. In the second it looks for a *method* named `length()` in `int[]` and again cannot find it. Another confusion is trying to access a string with

square brackets like an array. The form `s[0]` works in C and Python but does not compile in Java:

```
Arr.java:4: error: array required, but String found
        char c = s[0];
                 ^
```

An invalid index is not caught at compile time, because the value of an index is usually known only while the program runs. An index that goes beyond the string causes an exception at run time:

```
String s = "Java";
System.out.println(s.charAt(4));
```

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
    Index 4 out of bounds for length 4
```

`"Java"` has four characters; the valid indices are 0, 1, 2 and 3. The call `charAt(4)` throws `StringIndexOutOfBoundsException` and the program stops. The words “Index 4 out of bounds for length 4” in the message give the place of the problem exactly: index 4 was requested, while the length is 4. A negative index causes the same exception; for `"Java".charAt(-1)` the message is “Index -1 out of bounds for length 4”.

Common mistake

Trying to get the last character with `s.charAt(s.length())`. The length is one more than the last index, and this call throws `StringIndexOutOfBoundsException` on every string. The same error is made in a loop with the condition `i <= s.length()`. The correct condition is `i < s.length()`.

3.4.1.5 Looping over characters

To look at all the characters of a string one by one, a `for` loop is used. The loop variable starts at 0 and stops before reaching the length. The program in the slides takes a word from the user and prints the letters of that word at odd positions; the loop variable increases by 2 on every pass:

```
Scanner scanner = new Scanner(System.in);
System.out.print("Enter a word: ");
String word = scanner.nextLine();
for (int i = 0; i < word.length(); i += 2) {
  char letter = word.charAt(i);
  System.out.println((i + 1) + ". letter: " + letter);
}
scanner.close();
```

The program reads a line from the keyboard with `Scanner`. To use `Scanner`, the line `import java.util.Scanner;` is added at the top of the file; the details of the reading methods are in the user input section. Unlike `println`, `print` does not move to the next line. That is why the word the user types appears on the same line as the prompt. When the user types `Giresun` and presses `Enter`, the output is as follows:

```
Enter a word: Giresun
1. letter: G
3. letter: r
5. letter: s
7. letter: n
```

Table 3.4.2 gives every pass of the loop. `"Giresun"` has seven characters; its indices run from 0 to 6. The loop variable takes the values 0, 2, 4, 6. So that the ordinal number written on the screen starts at 1, as people are used to, `i + 1` is printed. The parentheses here are needed: `(i + 1) + ". letter: "` does the addition first. In this line the result would be the same without the parentheses, because the expression starts with a number. Still, the parentheses show clearly that the addition is done before the concatenation. In an expression that starts with a string, such as `"Letter " + i + 1`, writing it without parentheses produces `Letter 01` instead of `Letter 1` when `i` = 0.

Pass	`i`	`i < 7`	`charAt(i)`	Line written
1	0	true	`'G'`	`1. letter: G`
2	2	true	`'r'`	`3. letter: r`
3	4	true	`'s'`	`5. letter: s`
4	6	true	`'n'`	`7. letter: n`
—	8	false	—	the loop ends

Table 3.4.2: Tracing the loop for `word = "Giresun"` (length 7).

The last row is important. When `i` is 8, the condition is false and `charAt(8)` is never called. Because the condition is written as `i < word.length()`, the loop stops safely at any length. If the word had an even length, for example `"Ordu"`, the loop would read indices 0 and 2 and end at `i` = 4. If an empty line is entered, the length is 0 and the body never runs.

At the end of the loop `scanner.close()` is called. This call releases the resource used by the `Scanner`. When a `Scanner` that reads from the keyboard is closed, `System.in` is closed too, and the rest of the program can no longer read from the keyboard. That is why `close()` is called only after all the reading is done.

3.4.2 String methods

The `String` class offers ready-made methods for most everyday jobs with text: searching for a word, taking a part of the text, making letters uppercase, removing spaces, splitting the text into pieces. All these methods are called with a dot on a string object: `s.length()`, `s.indexOf("a")`. They also share one more property: none of them changes the string it is called on. The result always comes back as a new value.

3.4.2.1 Basic methods

Table 3.4.3 shows the most frequently used methods on the string `s = "Java Programming"`. This string has 16 characters; the space at index 4 counts as a character too. All the results in the table were obtained by running with JDK 21.

Method	What does it do?	Example and result
`length()`	Returns the number of characters.	`s.length()` → 16
`charAt(i)`	Returns the character at index `i` (`char`).	`s.charAt(0)` → `'J'`
`indexOf(str)`	The index where the substring first starts; −1 if it is not found.	`s.indexOf("Programming")` → 5 `s.indexOf("C++")` → −1
`indexOf(str, from)`	Starts the search at index `from`.	`s.indexOf("a", 4)` → 10
`lastIndexOf(str)`	The index of the last occurrence.	`s.lastIndexOf('a')` → 15
`substring(i1, i2)`	The substring in between, with `i1` included and `i2` excluded.	`s.substring(5, 12)` → `"Program"`
`substring(i1)`	From `i1` to the end.	`s.substring(5)` → `"Programming"`
`toLowerCase()`	A new lowercase string.	`"java programming"`
`toUpperCase()`	A new uppercase string.	`"JAVA PROGRAMMING"`

Table 3.4.3: Basic string methods, for `s = "Java Programming"`.

Definition 3.4.4

A **substring** is a part of a string made of consecutive characters. `s.substring(a, b)` returns the part of `s` that starts at index `a` and ends at index `b - 1`. The length of the part is `b - a`.

`indexOf` is a search method. It scans the string from start to end and gives the *starting* index of the first occurrence of the text sought. If the text does not occur at all, it returns −1. Because −1 cannot be a valid index, it carries the meaning "not found" safely. That is why the result of a search is often checked before use, in the form `if (s.indexOf(x) != -1)`. The two-parameter version of the method starts the search from a given index. The call `s.indexOf("a", 4)` skips the `a`s at 1 and 3 and first finds the `a` at index 10. `lastIndexOf`, on the other hand, scans the string from end to start and gives the last match. `indexOf` and `lastIndexOf` accept both a string and a single character (`'a'`).

The most confused aspect of `substring` is that the end index is *excluded*. This rule has two practical benefits. First, the length of the part is the difference of the two indices: `substring(5, 12)` gives 12 − 5 = 7 characters. Second, consecutive parts fit side by side easily: `s.substring(0, k) + s.substring(k)` gives `s` itself for every `k`. The one-parameter `substring(5)` is the same as `substring(5, s.length())`.

If an invalid range is given, `substring` also throws `StringIndexOutOfBoundsException`. The end cannot be greater than the length, and the start cannot be greater than the end. The message for the call `"Java".substring(2, 6)` gives the range in the half-open interval notation of mathematics:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
    Range [2, 6) out of bounds for length 4
```

The notation `[2, 6)` in the message means "2 included, 6 excluded". This notation also summarizes the rule of `substring`. On the other hand, `s.substring(s.length())` is valid and returns the empty string. Equal start and end mean an empty range.

3.4.2.2 indexOf and substring step by step

The trace slide in the slides runs the methods in the table in order and shows the value each variable gets on the character boxes:

```
String str = "Java Programming";
int index = str.indexOf("Programming");
int length = str.length();
String lo = str.substring(5, 12);
String last = str.substring(5);
String lower = str.toLowerCase();
String upper = str.toUpperCase();
```

Nothing is printed in the code; what matters is the values the variables get after each line. Table 3.4.4 gives these values in order.

Line	Variable	Value	Explanation
2	`index`	5	`"Programming"` starts at index 5
3	`length`	16	indices 0 … 15
4	`lo`	`"Program"`	indices 5 … 11; 12 − 5 = 7 characters
5	`last`	`"Programlama"`	from index 5 to the end
6	`lower`	`"java programming"`	a new string; `str` has not changed
7	`upper`	`"JAVA PROGRAMMING"`	a new string; `str` has not changed

Table 3.4.4: The value produced after each line on `str = "Java Programming"`.

Figure 3.4.4 shows two important steps with a drawing. The highlighted boxes are the characters the method reaches. In the `substring(5, 12)` step box number 12 (`g`) is not highlighted; the fact that the end index is excluded is visible in the drawing as well.

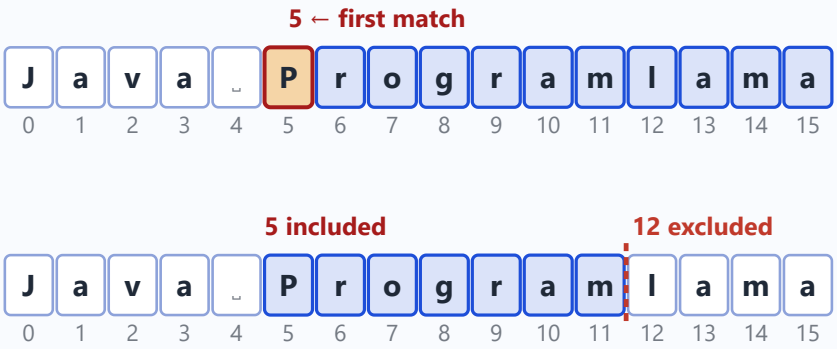


Figure 3.4.4: (a) `indexOf("Programming")` gives the index where the match starts, 5. (b) `substring(5, 12)` takes indices 5–11: `"Program"`.

In the last two lines `toLowerCase()` and `toUpperCase()` are called. Both return a new string, and this string is assigned to another variable. The variable `str` stays `"Java Programming"` throughout the program. The impression that a method "changes the string" is misleading; the reason is explained in the immutability section.

Example 3.4.1

Let us split an e-mail address into a user name and a domain name. Let the address be `e = "student@giresun.edu.tr"`. The position of the `@` sign is found with `int at = e.indexOf('@');`. `"student"` has seven letters; the letters occupy indices 0–6 and `@` is at index 7. The user name is the part before the `@`: `e.substring(0, at) = e.substring(0, 7) = student`. Because the end index is excluded, the `@` does not enter the part. The domain name is the part after the `@`: `e.substring(at + 1) = e.substring(8) = giresun.edu.tr`.

Had the `+ 1` been forgotten here, the result would have been `@giresun.edu.tr`. The method is independent of the length of the address; the indices are not counted by hand but found with `indexOf`.

3.4.2.3 Other useful methods

Besides the basic methods, a few more methods are frequent in everyday code. The example in the slides tries them together:

```
String s = "  Hello World  ";
System.out.println("[" + s.trim() + "]");
System.out.println("[" + s.replace('o', '0') + "]");
String[] fruits = "apple,pear,plum".split(",");
System.out.println(fruits.length);
System.out.println(fruits[1]);
System.out.println(String.join(" - ", fruits));
System.out.println("ab".repeat(3));       // Java 11+
System.out.println("".isEmpty());
System.out.println("   ".isBlank());      // Java 11+
char[] arr = "Java".toCharArray();
System.out.println(arr.length + " " + arr[0]);
System.out.println("Giresun".replace("sun", "SUN"));
System.out.println(String.valueOf(3.5) + 1);
System.out.println(Integer.parseInt("42") + 1);
```

```
[Hello World]
[  Hell0 W0rld  ]
3
pear
apple - pear - plum
ababab
true
true
4 J
GireSUN
3.51
43
```

Let us explain the lines in order:

- `trim()` removes the spaces at the start and end of the string. It does not touch the spaces in between: `"  Hello World  "` → `[Hello World]`. The square brackets were added to make the spaces visible. Calling `trim()` before processing user input is a common habit. `strip()`, which came with Java 11, does the same job for all the whitespace characters in Unicode.

- `replace('o', '0')` turns all the `o` letters into `0`: `[ Hell0 W0rld ]`. It changes not only the first match but all of them. The spaces are preserved, because `replace` produces the new string from `s` itself. The result of `trim()` on the previous line was not assigned anywhere.
- `split(",")` splits the string at the given separator and returns the pieces as a `String[]` array. `"apple,pear,plum"` is split into three pieces; `fruits.length` is 3 and `fruits[1]` is `pear`.
- `String.join(" - ", fruits)` does the opposite of splitting: it joins the pieces with the separator between them. The result is `apple - pear - plum`.
- `"ab".repeat(3)` writes the string three times side by side: `ababab`. This method was added in Java 11.
- `isEmpty()` returns `true` if the length is 0. `isBlank()` (Java 11) returns `true` if the string is empty or consists only of whitespace characters. `" "` is not empty but consists of nothing except spaces: `isEmpty()` gives `false` and `isBlank()` gives `true`.
- `toCharArray()` copies the characters of the string into a new `char` array. The length of the array is 4 and its first element is `'J'`. The array can be changed; so jobs such as swapping characters can be done on the array and the result converted back with `new String(arr)`.
- `replace` is not limited to a single character; it also takes strings: `"Giresun".replace("sun", "SUN")` → `GireSUN`.
- The last two lines show the conversion between numbers and strings. `String.valueOf(3.5)` is a string, `"3.5"`; adding 1 to it is concatenation and the result is `3.51`. `Integer.parseInt("42")` converts the string into a real `int`; when 1 is added, 43 comes out. For floating-point numbers `Double.parseDouble` is used.

`parseInt` accepts only a string made of digits (and an optional sign at the start). If a text containing spaces or letters is given, it throws `NumberFormatException`:

```
Exception in thread "main" java.lang.NumberFormatException:
    For input string: " 12 "
```

That is why number text coming from a user or a file is first cleaned with `trim()`: `Integer.parseInt(text.trim())`. This step matters even more when used together with splitting. When a line such as `" 3, 12 "` is split at the comma, spaces are left at the start and end of the pieces.

`split` has two more subtleties. An empty piece is produced between two consecutive separators. Empty pieces at the end of the string, however, are dropped. Also, the parameter of `split` is a **regular expression**: a pattern language used in text searching. The pattern `" +"` means "one or more spaces":

```
System.out.println(Arrays.toString("a b  c".split(" ")));
System.out.println(Arrays.toString("a b  c".split(" +")));
System.out.println(Arrays.toString("a,,b,".split(",")));
```

```
[a, b, , c]
[a, b, c]
[a, , b]
```

In the first line two consecutive spaces produce an empty piece and the array has four elements. The second line counts several spaces as a single separator. In the third line the empty piece in the middle remains and the one at the end is dropped; the array has three elements. The dot (`.`) and the vertical bar (`|`) have special meanings in regular expressions. That is why `"a.b".split(".")` does not give the expected result; to split at the dot, `split("\\.")` is written.

3.4.3 Immutability

In the earlier sections it was stressed several times that every method returns a new value. This is not a coincidence but the basic design decision of the `String` class. This section explains what the decision means, how it looks in memory, and the cost it brings. The pitfalls of case conversion in Turkish texts and the `StringBuilder` class, designed for large numbers of concatenations, are also in this section.

3.4.3.1 An immutable object

Definition 3.4.5

An **immutable** object is an object whose state never changes after it is created. `String` objects are immutable: the characters and the length of a string are fixed at the moment the object is created and never change afterwards. `toUpperCase` , `replace` , `substring` , `concat` and `+` do not touch the original object; they return the result as a new object.

An analogy helps. A string is like a stone tablet. The writing on the tablet cannot be scraped off and corrected. If a change is needed, a new tablet is written and put in place of the old one. The example in the slides shows this behavior:

```
String s = "Giresun Üniversitesi";
s.toUpperCase();      // the result is lost
System.out.println(s);
s = s.toUpperCase();  // the result is assigned to s
System.out.println(s);
```

```
Giresun Üniversitesi
GİRESUN ÜNİVERSİTESİ
```

The call `s.toUpperCase();` on the second line runs and produces a new uppercase string. But the result is assigned to no variable and is lost. That is why the first `println` prints the original text. On the fourth line the result is assigned to `s`. Here too the old object does not change; the variable `s` now points to the new object. What changes is not the object but the place the variable points to.

Figure 3.4.5 gives these two cases with a memory drawing. The stack in the left part holds the local variables of the method; `s` sits here. The **heap** on the right is the region where objects live. The arrow leaving the variable `s` points to the object it refers to.

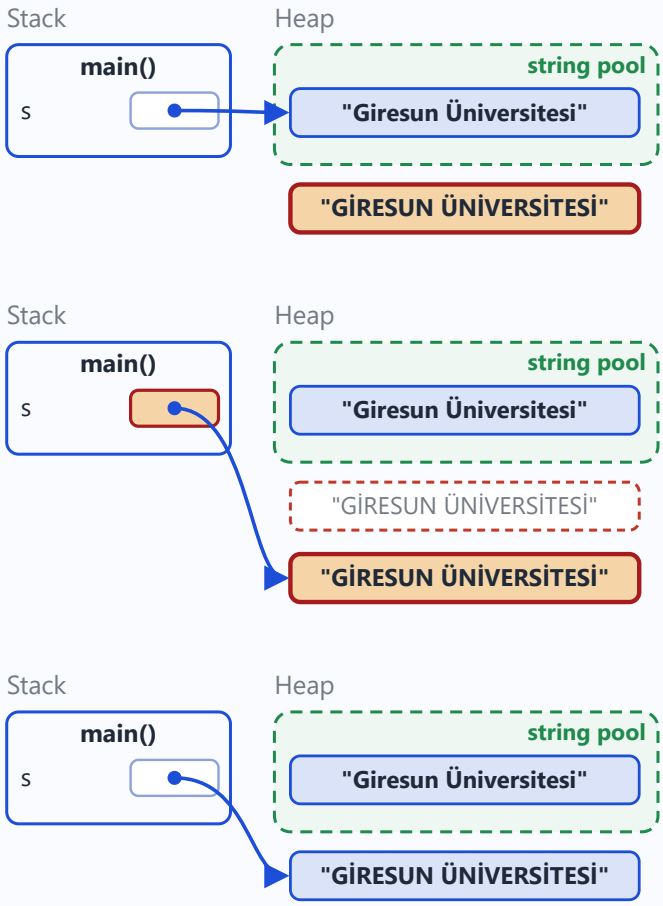


Figure 3.4.5: Immutability in memory. (a) `toUpperCase()` produces a new object, but because the result is not assigned, `s` keeps pointing to the old object. (b) The second call again produces a new object and this time its reference is assigned to `s`. (c) `s` now points to the uppercase object; the old object stays unchanged.

No variable points to the object produced by the first call. Such objects, which nobody points to, are left to the **garbage collector**. The garbage collector is the JVM component that reclaims the memory of objects no reference points to when the time comes. The programmer does not have to release this memory by hand.

The `GİRESUN ÜNİVERSİTESİ` on the second line of the output is written with the dotted capital `İ` letters. This result is obtained on a computer with the Turkish locale. On a computer with the English locale the same program prints `GIRESUN ÜNIVERSITESI`. The reason is explained in the next subsection.

Immutability has a price: every change means a new object. The designers of Java accepted this price for the following benefits:

- **Safe sharing.** Two variables pointing to the same string cannot corrupt each other. There is no need to fear that a string given to a method will be changed inside that method.
- **The string pool.** String literals with the same content can share a single object. Sharing is safe only if the object does not change. The pool is treated in detail in the comparison section.
- **Safety between threads.** Parts of a program that run at the same time can share an object that does not change without taking extra precautions.

Common mistake

Forgetting to assign the result of a string method: the lines `name.trim();`, `name.toUpperCase();` or `name.replace(" ", "");` do not change `name`. The compiler gives no warning; the program runs but nothing happens. The correct form is `name = name.trim();`. For the same reason an attempt to change a character in place, such as `s.charAt(0) = 'j';`, does not compile.

3.4.3.2 Turkish letters and the locale

Definition 3.4.6

A **locale** is the set of format rules specific to a language and a region: case conversion, decimal separator, thousands separator, date format. In Java it is represented by the class `java.util.Locale`. The locale for Turkish as spoken in Turkey is `tr-TR`.

The Turkish alphabet has two forms of the letter `i`, dotted and dotless, and each has its own uppercase: `i ⇔ İ`, `ı ⇔ I`. In English there is only the pair `i ⇔ I`. The parameterless `toUpperCase()` and `toLowerCase()` use the default locale of the computer. The same program can therefore give different results on different machines:

```
import java.util.Locale;

String s = "Giresun Üniversitesi";
System.out.println(s.toUpperCase());
System.out.println(s.toUpperCase(Locale.ROOT));
System.out.println("TITLE".toLowerCase());
System.out.println("TITLE".toLowerCase().equals("title"));
System.out.println("TITLE".toLowerCase(Locale.ROOT));
```

```
GİRESUN ÜNİVERSİTESİ
GIRESUN ÜNIVERSITESI
tıtle
false
title
```

The output was obtained on a computer set to Turkish. In the first line, since the default locale is Turkish, the `i` letters become `İ`. In the second line `Locale.ROOT` is given. This is the language-independent locale that uses the basic Unicode rules; the `i` letters become `I`. The third line shows the trap: when `"TITLE"` is lowercased with Turkish rules, the `I` letters become `ı` and the result is `tıtle`. The comparison on the fourth line therefore gives `false`. On the last line the expected `title` is obtained with `Locale.ROOT`. Table 3.4.5 summarizes the conversion of four letters in the two locales.

Conversion	Turkish (`tr`)	`ROOT` / English
`i` → uppercase	`İ`	`I`
`I` → lowercase	`ı`	`i`
`ı` → uppercase	`I`	`I`
`İ` → lowercase	`i`	`i` (`i` + combining dot; two characters)

Table 3.4.5: Case conversion in the Turkish (`tr`) and language-independent (`Locale.ROOT`) locales.

The last row of the table is a less well-known case. When `İ` is lowercased with non-Turkish rules, the result is not a single `i` but an `i` followed by a “combining dot” character. There is no visible difference, but the length of the string increases. `"İSTANBUL".toLowerCase(Locale.ROOT)` gives a string of 9 characters; `"İki".toLowerCase(Locale.ROOT).equals("iki")`, on the other hand, is `false`.

The practical rule is this. In texts inside the program, that is, texts that a machine will read such as command names, file extensions and protocol keys, `Locale.ROOT` is used. In Turkish texts to be shown to the user, the Turkish locale is given explicitly:

`text.toUpperCase(Locale.forLanguageTag("tr-TR"))`. This way the program produces the same result on whatever computer it runs. The `Locale` class is in the `java.util` package and needs the line `import java.util.Locale;`.

Common mistake

Checking user input with `input.toLowerCase().equals("exit")`. On a computer set to Turkish, if the user types `EXIT`, the result is `"exıt"` and the condition does not hold. Because the error appears only on some machines, it is hard to find. To ignore the difference between uppercase and lowercase, `equalsIgnoreCase` or `toLowerCase(Locale.ROOT)` is used.

3.4.3.3 The cost of concatenating in a loop

The price of immutability is felt most when concatenating in a loop. The first part of the code in the slides collects the numbers from 1 to 5 in a string:

```
String result = "";
for (int i = 1; i <= 5; i++) {
  result += i + " ";   // a new String on every pass
}
System.out.println(result);

StringBuilder sb = new StringBuilder();
for (int i = 1; i <= 5; i++) {
  sb.append(i).append(' ');   // the same object grows
}
String text = sb.toString();
System.out.println(text);

StringBuilder rev = new StringBuilder("Giresun");
rev.reverse();
System.out.println(rev);
StringBuilder k = new StringBuilder("Java");
k.insert(0, ">> ").append('!');
k.setCharAt(3, 'j');
System.out.println(k + " " + k.length());
```

```
1 2 3 4 5
1 2 3 4 5
nuseriG
>> java! 8
```

In the first loop the line `result += i + " ";` produces a new `String` object on every pass. The old content and the new piece are copied into this object, and the old object is left as garbage. The number of characters copied in one pass equals the length of the text accumulated so far. If each pass adds one character, the total number of characters copied is $1 + 2 + \ldots + n = n(n + 1)/2$, that is, about $n^2/2$. For example, in a loop of a thousand passes 500,500 characters are copied. This growth is written $O(n^2)$: when the number of passes doubles, the work quadruples.

In this example of five passes the difference is not felt. Each pass adds two characters (a digit and a space); a total of 2 + 4 + 6 + 8 + 10 = 30 characters are copied. But if a report of tens of thousands of lines or the contents of a large file are built this way, the program slows down noticeably.

3.4.3.4 StringBuilder

Definition 3.4.7

`StringBuilder` is a **mutable** character **buffer**. It contains an array that holds the characters and a length that shows how many cells of this array are filled. Methods such as `append`, `insert`, `reverse`, `setCharAt` and `deleteCharAt` do not produce a new object; they change the contents of the same object. When the job is done, a `String` is obtained with `toString()`.

The second part of the code builds the same text with `StringBuilder`. `sb.append(i).append(' ')` adds a number and a space to the end of the buffer on every pass. The buffer is a single object and grows as needed. When the loop ends, `sb.toString()` produces a `String` from the contents. The output of the two methods is the same: `1 2 3 4 5 ` (with a space at the end).

The `append` method returns the buffer itself. That is why the calls can be written one after another like a chain: `sb.append(i).append(' ')`. The first `append` returns `sb`, and the second `append` again works on `sb`. `append` has a separate version for each primitive type and for `String`; numbers, characters and booleans are all converted to text and appended.

The last part of the code tries the other methods. `new StringBuilder("Giresun")` creates a buffer whose content is `"Giresun"`. `reverse()` reverses the buffer in place and `nuseriG` is printed. When a `StringBuilder` is given to `println`, `toString()` is called automatically. In the last example three characters are inserted at the start of the `"Java"` buffer with `insert(0, ">> ")`, `'!'` is put at the end, and the `J` at index 3 is turned into `j` with `setCharAt`. The result is `>> java!` and the length is 8. This operation is not possible with `String`; to change a character, a new string has to be built every time.

The trace slide in the slides shows the inside of the buffer cell by cell:

```
StringBuilder sb = new StringBuilder("Java");
sb.append(' ').append(8);
sb.insert(0, ">> ");
sb.setCharAt(3, 'j');
sb.deleteCharAt(sb.length() - 1);
sb.append("programming book");
String s = sb.toString();
System.out.println(s + " " + sb.length() + " " + sb.capacity());
```

```
>> java programming book 24 42
```

A `StringBuilder` has two measures. The **length** (`length()`) is the number of filled cells. The **capacity** (`capacity()`) is the total number of cells of the internal array. The capacity of a buffer created from a string is found by adding 16 to the length of the string. For `"Java"` this is 4 + 16 = 20. The capacity of an empty `new StringBuilder()` is 16. Table 3.4.6 gives the state after each line; the values were verified by printing `length()` and `capacity()`.

Line	Content	Length	Capacity	Explanation
`new StringBuilder("Java")`	`[Java]`	4	20	16 + 4
`append(' ').append(8)`	`[Java 8]`	6	20	appended at the end, the object is the same
`insert(0, ">> ")`	`[>> Java 8]`	9	20	the old characters shift 3 cells to the right
`setCharAt(3, 'j')`	`[>> java 8]`	9	20	index 3 is changed in place
`deleteCharAt(length() - 1)`	`[>> java ]`	8	20	the `8` at the end is deleted
`append("programming book")`	`[>> java programming book]`	24	42	24 > 20: the buffer grows
`toString()`	—	24	42	a new `String` from the contents

Table 3.4.6: `StringBuilder` trace. The text in square brackets is the filled part of the buffer.

`insert` and `deleteCharAt` are not cheap operations. To insert something at the start, all the following characters must be shifted to the right; deleting from the middle also requires shifting to the left. These shifts do work proportional to the length of the buffer. `append`, on the other hand, writes only at the end as long as there is room, and finishes in constant time.

The last `append` adds 16 characters and the length rises to 24. 24 characters do not fit into 20 cells. In this case `StringBuilder` allocates a new, larger array, copies the old contents into it, and abandons the old array. The rule in the JDK chooses the new capacity as twice the old one plus two: 2 × 20 + 2 = 42. The notation *(20 + 1) × 2* in the slides gives the same number. If the required length is greater than that, that length is taken directly. Figure 3.4.6 shows the moment of growth.

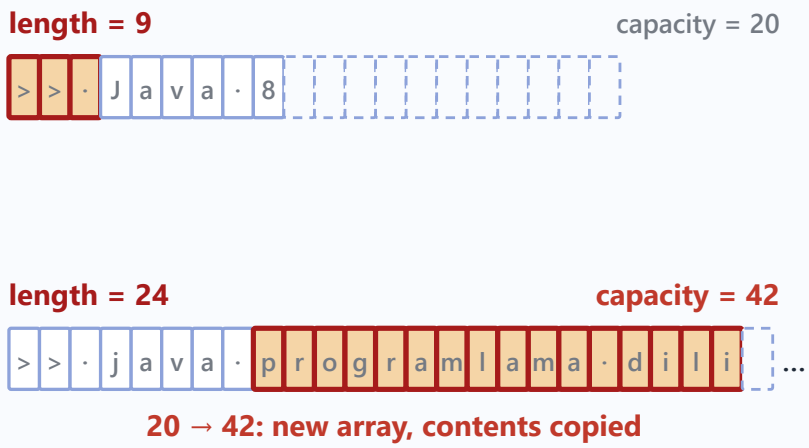


Figure 3.4.6: The `StringBuilder` buffer. (a) After `insert(0, ">> ")` : the old characters have shifted three cells to the right. (b) 24 characters do not fit into 20 cells; the capacity rises to 42 and the contents are copied to the new array.

Why is `StringBuilder` fast although copying takes place during growth? The capacity roughly doubles at every growth. When characters are added one by one to an empty buffer, the capacity goes 16, 34, 70, 142, 286, 574, 1150. Only six growths are needed to add a thousand characters. The total number of characters copied in these growths does not exceed a few times the number of characters added. So the total cost of adding n characters is $O(n)$; compared with the $O(n^2)$ cost of concatenation with `+=`, the difference is very large.

This does not mean that the `+` operator should never be used. For concatenations of a few pieces `+` is readable and fast enough; the Java compiler already translates the concatenations in a single expression efficiently. `StringBuilder` is preferred when many pieces are joined inside a loop. If the approximate length is known in advance, giving an initial capacity such as `new StringBuilder(1000)` reduces the number of growths.

3.4.4 User input: Scanner

A program often asks the user for information: a name, an age, a sentence. Everything that comes from the keyboard first comes as characters. Converting these characters into a word, a line or a number is the job of the `Scanner` class. Although the reading methods look alike, their behavior differs in subtle points. If these distinctions are not known, programs produce odd bugs such as skipping the user's question.

3.4.4.1 Scanner and the reading methods

Definition 3.4.8

`java.util.Scanner` is a class that reads the text in a source by splitting it into pieces. `new Scanner(System.in)` creates an object that reads from the keyboard. The line typed at the keyboard enters an **input buffer** when `Enter` is pressed; the newline character (⏎) is also added to the buffer. `Scanner` keeps a **cursor** in the buffer, and each read moves the cursor to the end of the part that was read.

Four reading methods are frequently used:

- `next()` first skips the leading whitespace (space, tab, newline) and then reads up to the next whitespace. The result is a single word.
- `nextLine()` reads everything from where the cursor is to the end of the line. It does not skip spaces. It consumes the newline character but does not include it in the result.
- `nextInt()` and `nextDouble()` skip leading whitespace and read a number. They **leave** the newline after the number in the buffer. If a letter is typed instead of a number, they throw `InputMismatchException`.

A program that reads from the keyboard contains the line `import java.util.Scanner;` at the top of the file. When the reading is finished, `keyboard.close()` is called. The example in the slides reads one line first as two words and then as the rest of the line:

```
Scanner keyboard = new Scanner(System.in);
System.out.println("enter some text");
String input = keyboard.next();
System.out.println(input);    // algorithms
input = keyboard.next();
System.out.println(input);    // in
input = keyboard.nextLine();
System.out.println(input);    //  programming lessons
```

The user types the following line and presses `Enter`:

```
algorithms in programming lessons
```

The program prints:

```
enter some text
algorithms
in
 programming lessons
```

The leading space on the last line of the output is noticeable. The space after the word “in” is part of the line, and `nextLine()` does not skip spaces. That is why the text read is `" programming lessons"`.

3.4.4.2 next() and nextLine() step by step

Table 3.4.7 gives where the cursor moves from and to on each read. The indices of the characters in the buffer start at 0: `"algorithms"` occupies 0–9, the first space 10, `"in"` 11–12, the second space 13, the remaining text 14–32, and the newline is at 33.

Call	Cursor before	Read	Cursor after	Explanation
`next()`	0	`"algorithms"`	10	reads up to the first space; the space stays in the buffer
`next()`	10	`"in"`	13	first skips the space at 10
`nextLine()`	13	`" programming lessons"`	34	does not skip the space; consumes ⏎ and does not include it in the result

Table 3.4.7: The position of the cursor while reading the line `algorithms in programming lessons⏎`.

Figure 3.4.7 shows the state of the buffer at the end of the second and third reads. The characters that have been read are faded, and the place of the cursor is marked with a triangle.

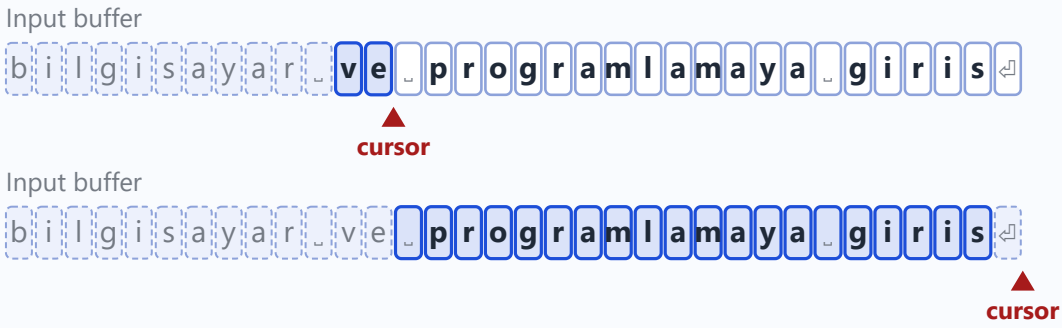


Figure 3.4.7: The input buffer. (a) The second `next()` skips the space and reads `"in"`; the cursor is again in front of a space. (b) `nextLine()` reads to the end of the line without skipping the space and consumes ⏎.

The result for the programmer is this: for a one-word answer (a first name, a city, a command) `next()` is used, and for an answer that may contain spaces (a full name, an address, a sentence) `nextLine()` is used. If a full name is read with `next()`, only the first name is taken and the surname stays in the buffer and gets mixed into the next read.

3.4.4.3 The nextLine() trap after nextInt()

When `nextLine()` is used right after the methods that read numbers, an error appears that almost every beginner meets. The program in the slides first asks for the age and then for the name:

```
Scanner keyboard = new Scanner(System.in);
System.out.print("Age: ");
int age = keyboard.nextInt();
System.out.print("Name: ");
String name = keyboard.nextLine();
System.out.println("[" + name + "] " + age);
```

The user types `19` and presses `Enter`. The program ends without asking for the name:

```
Age: 19
Name: [] 19
```

Why this happens can be understood by looking at the buffer (Figure 3.4.8). The buffer holds `19⏎`. `nextInt()` reads only `19` and leaves the cursor in front of the ⏎. The `nextLine()` that follows does its job: it reads from the cursor to the end of the line. Since there are no characters between the cursor and the newline, the result is the empty string. `nextLine()` consumes the ⏎ and returns at once. Before the user has had a chance to type a name, the variable `name` has become `""`.



Figure 3.4.8: After `nextInt()` reads 19, the cursor waits in front of the newline character.

The solution is to consume the leftover newline with a separate `nextLine()` call after reading the number:

```
int age = keyboard.nextInt();
keyboard.nextLine();          // discard the ⏎
System.out.print("Name: ");
String name = keyboard.nextLine();
```

```
Age: 19
Name: Ayşe
[Ayşe] 19
```

The result of the `keyboard.nextLine();` call inserted in between is not used; its only job is to clear the ⏎ from the buffer. The `nextLine()` after that waits for the user's new line. A second solution is to read all the input with `nextLine()` and convert the numbers yourself: `int age = Integer.parseInt(keyboard.nextLine().trim());`. With this method no leftover newline is ever left in the buffer.

Example 3.4.2

A program that asks the user first for the age and then for a full name and writes the result with `printf` is written like this, without falling into the trap:

```
Scanner k = new Scanner(System.in);
System.out.print("Age: ");
int age = k.nextInt();
k.nextLine();   // discard the newline
System.out.print("Name: ");
String name = k.nextLine();
System.out.printf("Name: %s, Age: %d%n", name, age);
```

When the user enters `20` and `Ali Veli`, the console looks like this:

```
Age: 20
Name: Ali Veli
Name: Ali Veli, Age: 20
```

`nextInt()` reads 20, and the ⏎ stays in the buffer. `k.nextLine();` consumes this ⏎. The last `nextLine()` waits for the user's new line and reads `"Ali Veli"` together with its space. Had `next()` been used for the full name, only `"Ali"` would have been read. The `printf` line is explained in the formatted output section.

3.4.5 Comparing strings

Asking whether two strings are the same looks simple. But in Java this question has two different meanings: “do the two variables point to the same object?” and “is the content of the two objects the same?”. The first question is asked with `==` and the second with `equals()`. This section explains the

difference between the two questions, the string pool that sometimes hides the difference, and the `compareTo` method used for ordering.

3.4.5.1 Comparing with ==

For primitive types such as `int`, `==` compares the two values. `String` variables, however, hold references, not values. So `==` compares the **references** of two strings, that is, their addresses in memory, not their contents. If two variables point to the same object, the result is `true`. Even if the contents are the same, if the objects differ the result is `false`. The example in the slides:

```
String str1 = "Giresun";
String str2 = "Giresun";
String str3 = new String("Giresun");
System.out.println(str1 == str2);        // true
System.out.println(str1 == "Giresun");   // true
System.out.println(str1 == str3);        // false
System.out.println(str3 == "Giresun");   // false
```

```
true
true
false
false
```

In all four comparisons the content is always `"Giresun"`, but two of the results are `false`. That `str1 == str2` comes out `true` is no luck; it is a consequence of the string pool. `str3` is a separate object created with `new`.

3.4.5.2 The string pool

Definition 3.4.9

The **string pool** is the special region where the JVM keeps string literals. All string literals with the same content in the source code share a single object in the pool. `new String(...)`, on the other hand, does not look at the pool; on every call it creates a new object on the heap.

The purpose of the pool is to save memory. The literal `"Giresun"` may occur in a hundred different places in a program. Instead of creating a new object at each occurrence, all of them point to the same object. This sharing is safe only because strings are immutable. If the code in one place could change the object `"Giresun"`, the other ninety-nine places would be affected too.

Table 3.4.8 gives the pool trace from the slides line by line. The objects are numbered #1 and #2.

Line	str1	str2	str3	Explanation
`String str1 = "Giresun";`	#1			the literal gives #1 in the pool
`String str2 = "Giresun";`	#1	#1		the same literal: no new object is created
`String str3 = new String("Giresun");`	#1	#1	#2	`new` produces a new object outside the pool
`str1 == str2`	#1 and #1			`true`
`str1 == "Giresun"`	#1 and #1			`true`
`str1 == str3`	#1 and #2			`false`
`str3 == "Giresun"`	#2 and #1			`false`

Table 3.4.8: The string pool trace. #1 is the object in the pool and #2 is the object outside the pool.

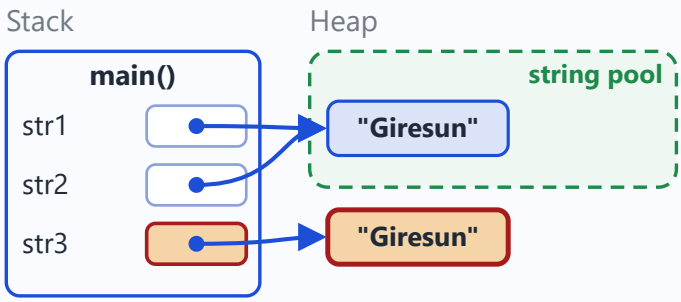


Figure 3.4.9: The memory after the three declarations. `str1` and `str2` point to the same object in the pool, and `str3` points to a separate object outside the pool.

Strings that enter the pool are not limited to the literals in the source code. Expressions that can be computed at compile time are also treated as constants. Both sides of the expression `"Gire" + "sun"` are constants; the compiler writes it directly as `"Giresun"` and the result points to the object in the pool. A concatenation with a variable on one side is done at run time and produces a new object. A variable defined with `final` and initialized with a constant is a constant for the compiler:

```
String str1 = "Giresun";
String a = "Gire";
final String fa = "Gire";
System.out.println((a + "sun") == str1);   // false
System.out.println((fa + "sun") == str1);  // true
```

The `intern()` method returns the pool counterpart of a string. If an object with the same content exists in the pool, the reference of that object is returned; if not, the string is added to the pool and itself is returned. `str3.intern() == str1` is therefore `true`. `intern()` is rarely needed in everyday code; its existence helps to understand how the pool works.

If you want to follow the pool and immutability step by step, you can use the String Pool simulator on the course site. The simulator visually shows how literals enter the pool, the separate objects created with `new String`, and the in-place change of a `StringBuilder` buffer.

3.4.5.3 Comparing content with equals()

Definition 3.4.10

`a.equals(b)` compares the contents of the strings `a` and `b` character by character. If the lengths and the character at every position are the same, it returns `true`. It does not look at whether the objects are the same. If `b` is `null`, it returns `false`.

The example in the slides compares the same three variables, this time with `equals`, and adds a few cases:

```
String str1 = "Giresun";
String str2 = "Giresun";
String str3 = new String("Giresun");
System.out.println(str1.equals(str2));
System.out.println(str1.equals("Giresun"));
System.out.println(str1.equals(str3));
System.out.println(str3.equals("Giresun"));
System.out.println(str3.intern() == str1);
String a = "Gire";
String b = a + "sun";  // new object
System.out.println(b == str1);
System.out.println(b.equals(str1));
```

```
true
true
true
true
true
false
true
```

The first four lines give `true` whichever way the object was created; the content is always `"Giresun"`. The fifth line shows that `intern()` returns the pool counterpart. In the last two lines `b` is formed by joining the variable `a` and the constant `"sun"` at run time. The resulting object is new: `b == str1` is `false` and `b.equals(str1)` is `true`.

In practice the most dangerous case is user input. A string read from a `Scanner` is created at run time and is always a new object. So the condition `input == "yes"` gives `false` almost always, even if the user types exactly `yes`. The bug is insidious: in experiments with string literals, `==` seems to work correctly because of the pool.

`equals` has a `null` side too. `null` is the reference value that points to no object. A method cannot be called through a `null` reference:

```
String answer = null;
System.out.println("yes".equals(answer)); // false
System.out.println(answer + "!");          // null!
System.out.println(answer.equals("yes")); // exception
```

In the first line the method is called on the constant `"yes"`; since a constant is never `null`, the call is safe and returns `false` with a `null` argument. In the second line concatenation turns `null` into the text `"null"`; no exception occurs and `null!` is printed. The third line throws `NullPointerException`. Since Java 15 the message of this exception says by default which call failed and why:

```
Exception in thread "main" java.lang.NullPointerException:
    Cannot invoke "String.equals(Object)" because "answer" is null
```

The variable name in the message (`"answer"`) is visible when the class file is compiled with debugging information. Development environments add this information by default. In a program compiled with only `javac` on the command line, the ordinal number of the local variable is written instead of the name:

```
Exception in thread "main" java.lang.NullPointerException:
    Cannot invoke "String.equals(Object)" because "<local1>" is null
```

When compiled with `javac -g`, the message shows the name `"answer"` again. Both messages describe the same error: `equals` was called on a variable that is `null`. The safe form is to put the constant first: `"yes".equals(answer)`.

3.4.5.4 Comparison methods

Besides `equals`, strings have a few more comparison methods. Table 3.4.9 lists them together with the type they return.

Method	Returns	Explanation
`equals(str)`	`boolean`	Is the content exactly the same?
`equalsIgnoreCase(str)`	`boolean`	Is it the same when the difference between uppercase and lowercase is ignored?
`startsWith(str)`	`boolean`	Does the string start with the given text?
`endsWith(str)`	`boolean`	Does the string end with the given text?
`contains(str)`	`boolean`	Does the given text occur inside the string?

Method	Returns	Explanation
`isEmpty()` / `isBlank()`	`boolean`	Is the length 0? / Does it consist only of whitespace? (`isBlank` : Java 11+)
`compareTo(str)`	`int`	Dictionary order: negative (comes first), 0 (equal), positive (comes after).
`compareToIgnoreCase(str)`	`int`	`compareTo` that ignores letter case.

Table 3.4.9: String comparison methods.

Methods whose name does not contain `IgnoreCase` are case-sensitive. `equals` and `equalsIgnoreCase` do not throw an exception when given a `null` argument; they return `false`. The example in the slides tries these methods on the string `"Giresun Üni."`; the results in the comments are the real output of the program:

```
String s = "Giresun Üni.";
System.out.println(s.startsWith("Gir"));                // true
System.out.println(s.startsWith("Gır"));                // false
System.out.println(s.endsWith("Üni."));                 // true
System.out.println(s.endsWith("Üni"));                  // false
System.out.println(s.equalsIgnoreCase("Giresun Üni.")); // true
System.out.println(s.equalsIgnoreCase("Gıresun üni.")); // true
System.out.println(s.equalsIgnoreCase("GIresun Uni.")); // false
System.out.println(s.contains("Üni"));                  // true
```

Some of the results are surprising at first sight:

- `startsWith("Gır")` is `false`. `ı` and `i` are separate characters; their Unicode codes are 305 and 105.
- `endsWith("Üni")` is `false`, because the string ends with a period. The last character is `'.'`.
- `equalsIgnoreCase("Gıresun üni.")` is `true`. When comparing characters, the method converts both sides to uppercase. The uppercase forms of `ı` and `i` are the same, `I`; so the two strings are treated as equal. `equalsIgnoreCase` does not look at the locale; it gives the same result on every machine.
- `equalsIgnoreCase("GIresun Uni.")` is `false`. `U` and `Ü` are separate letters, and no case conversion makes them equal.



Figure 3.4.10: The regions matched by the methods `startsWith`, `contains` and `endsWith`.

Comparisons with the empty string are also worth knowing. Every string starts with the empty string and contains it: `s.startsWith("")` and `s.contains("")` are always `true`. `s.indexOf("")` returns 0.

3.4.5.5 compareTo and dictionary order

Definition 3.4.11

`a.compareTo(b)` compares two strings in **lexicographic order**. It returns a negative `int` if `a` comes first, 0 if they are equal, and a positive one if it comes after. The comparison starts at the beginning and ends at the first differing character; the result is the difference of the Unicode codes of these two characters. If there is no differing character and one is a prefix of the other, the result is the difference of the lengths.

The example in the slides contains all the cases of the rule:

```
System.out.println("egg".compareTo("apple"));
System.out.println("apple".compareTo("egg"));
System.out.println("egg".compareTo("egg"));
System.out.println("Egg".compareTo("egg"));
System.out.println("Egg".compareToIgnoreCase("egg"));
System.out.println("do".compareTo("door"));
System.out.println("çay".compareTo("dut"));
```

```
4
-4
0
-32
0
-2
131
```

Let us read the lines according to the rule:

- `"egg".compareTo("apple")`: the first characters differ. `'e'` − `'a'` = 101 − 97 = **4**. Positive: “egg” comes after “apple”.
- `"apple".compareTo("egg")`: the same characters in reverse order, **−4**. When the two are swapped, the sign changes.
- Equal strings give **0**.
- `"Egg".compareTo("egg")`: `'E'` − `'e'` = 69 − 101 = **−32**. In Unicode, uppercase letters come before lowercase letters.

- `compareToIgnoreCase` ignores letter case and gives **0**.
- `"do".compareTo("door")`: the first two characters are the same and `"do"` ends. One is a prefix of the other; the result is the difference of the lengths: 2 − 4 = **−2**.
- `"çay".compareTo("dut")`: the code of `'ç'` is 231 and the code of `'d'` is 100. 231 − 100 = **131**. A positive result means that "çay" comes after "dut".

The last line shows an important limit: `compareTo` does not know the Turkish alphabet order. In Turkish, `ç` lies between `c` and `d`. But in Unicode the codes of the letters specific to Turkish are larger than those of the Latin letters: `ç` is 231, `ı` 305, `ş` 351, `ü` 252, while `z` is 122. In a list of Turkish words sorted in Unicode order, words that start with `ç`, `ı` or `ş` fall after `z`. For correct Turkish sorting, `java.text.Collator` is used:

```
import java.text.Collator;
import java.util.Arrays;
import java.util.Locale;

String[] m = {"zeytin", "çilek", "Elma", "armut", "ıhlamur", "incir"};
String[] m2 = m.clone();
Arrays.sort(m);                          // Unicode order
System.out.println(Arrays.toString(m));
Collator tr = Collator.getInstance(Locale.forLanguageTag("tr-TR"));
Arrays.sort(m2, tr);                     // Turkish alphabet order
System.out.println(Arrays.toString(m2));
```

```
[Elma, armut, incir, zeytin, çilek, ıhlamur]
[armut, çilek, Elma, ıhlamur, incir, zeytin]
```

`Arrays.sort` sorts strings with `compareTo`. As a result "Elma", which starts with a capital letter, falls at the very front, and "çilek" and "ıhlamur" fall at the very end. The Turkish `Collator`, on the other hand, applies the alphabet order and puts the difference between uppercase and lowercase in the background when sorting. For lists to be shown to the user, the second way is chosen.

The trace slide in the slides shows the steps of `compareTo` on three pairs of strings:

```
String a = "case", b = "cave";
int r1 = a.compareTo(b);
int r2 = "car".compareTo("cargo");
int r3 = "Zeki".compareTo("ali");
System.out.println(r1 + " " + r2 + " " + r3);
```

```
-3 -2 -7
```

Pair	k	Characters (codes)	Case
"case", "cave"	0	'm' (109), 'm' (109)	equal, continue
	1	'a' (97), 'a' (97)	equal, continue
	2	's' (115), 'v' (118)	different: 115 − 118 = **−3**
"car", "cargo"	0–2	all equal; "car" ends	prefix: 3 − 5 = **−2**
"Zeki", "ali"	0	'Z' (90), 'a' (97)	different: 90 − 97 = **−7**

Table 3.4.10: `compareTo` trace. `k` is the index being compared.

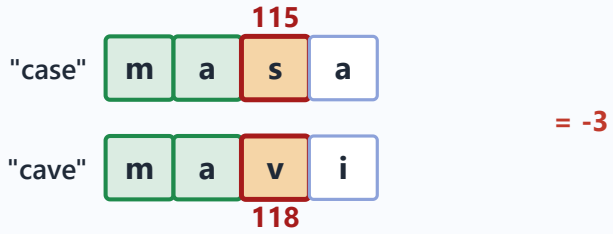


Figure 3.4.11: `"case"` and `"cave"` differ at index 2; the result is the difference of the codes, 115 − 118 = −3.

The third pair shows how far the uppercase rule goes: even `Z` comes before `a`. The size of the result usually carries no meaning; what matters is its sign. That is why the comparison is written `if (a.compareTo(b) < 0)` and not `== -1`. The operators `<` and `>` do not work with strings; the compiler gives a “bad operand types” error:

```
Compare.java:4: error: bad operand types for binary operator '<'
        if (a < b) System.out.println(a);
             ^
  first type:  String
  second type: String
```

3.4.6 Formatted output

`println` writes a value as it is. When a grade sheet, an invoice or a table of measurements is printed, this is not enough. Columns must be aligned and floating-point numbers written with a certain number of digits. For this job Java offers the `printf` method and `String.format`, which uses the same rules.

3.4.6.1 printf and format specifiers

Definition 3.4.12

`System.out.printf(format, arguments…)` takes a **format string** followed by any number of arguments. The format string contains **format specifiers** placed among plain text. Each specifier starts with `%` and ends with a type letter; the optional part in between gives the flag, the width and the precision. The arguments take the place of the specifiers in order.

The general form of a specifier is `%[flag][width][.precision]type` (Figure 3.4.12). In the specifier `%-8.2f`, the `-` flag left-aligns, 8 is the width, `.2` is the precision and `f` is the type.



Figure 3.4.12: The parts of a format specifier: start (`%`), flag, width, precision and type.

Table 3.4.11 gives the most frequently used type letters. The decimal separator depends on the locale. A comma is used with the Turkish locale and a point with `Locale.US`.

Specifier	Type	Example → output
`%d`	Integer (`byte` , `short` , `int` , `long`)	42 → 42
`%f`	Floating-point (`float` , `double`)	`%.1f` : 3.14159 → `3,1`
`%s`	String (every value is converted to text)	`%5s` : `"ab"` → ␣␣␣ab
`%c`	Character	`'Z'` → Z
`%b`	Boolean	`true` → `true`
`%n`	Line break	—

Table 3.4.11: Basic format specifiers (outputs with the `tr-TR` locale).

`printf` does not move to the next line by itself. For a line break, `\n` or `%n` is added at the end of the format string. `%n` uses the line-break character of the operating system; on Windows it writes the two characters `\r\n`, on Linux and macOS `\n`. That is why `%n` is preferred in portable code.

The width is the minimum number of characters the value will occupy. If the value is shorter, it is padded with spaces: `%Wd` right-aligns and `%-Wd` left-aligns. If the value is longer than the width, it is not cut; the field grows. The precision is the number of digits after the decimal point for floating-point numbers, and the last digit is rounded: `%.2f` writes two digits. `%8.2f` sets aside a field of 8 characters in total, including the two decimal digits.

3.4.6.2 Width and precision

The example in the slides writes the same three values in four different formats:

```
int i = 3;
float f = -17;
String s = "java";

System.out.printf("x: %d y: %f s: %s\n", i, f, s);
System.out.printf("x: %4d y: %5.2f s: %6s\n", i, f, s);
System.out.printf("x: %-4d y: %-7.2f s: %-6s\n", i, f, s);
System.out.printf("x: %-4d y: %-6.3f s: %-6s\n", i, f, s);
```

```
x: 3 y: -17,000000 s: java
x:    3 y: -17,00 s:   java
x: 3    y: -17,00  s: java··
x: 3    y: -17,000 s: java··
```

The output was obtained with the Turkish locale. The spaces at the end of the lines are shown with `·` so that they are visible. Two of the variables need an explanation: `f` is a `float`, and when the integer `-17` is assigned to it, it becomes -17.0. If no precision is given to `%f`, 6 decimal digits are written. Let us examine the lines in order:

1. No width is given. Each value takes as much room as its own length: `3`, `-17,000000`, `java`.
2. `%4d` right-aligns 3 in a field of four characters: three spaces and 3. `%5.2f` writes `-17,00` with two decimals; this is 6 characters and does not fit in a width of 5, so the field grows. `%6s` puts two spaces to the left of the four-letter `java`.
3. The minus flag left-aligns; spaces are added to the right of the value. `%-4d` → `3···`, `%-7.2f` → `-17,00·`, `%-6s` → `java··`.
4. `%-6.3f` writes `-17,000` with three decimals. The value is 7 characters long and longer than the width of 6; no padding is done.

Figure 3.4.13 shows the second line character by character. The field occupied by each specifier is marked at the bottom.

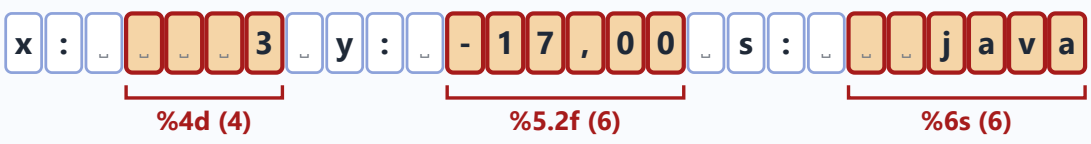


Figure 3.4.13: The output of `"x: %4d y: %5.2f s: %6s"` character by character. `%4d` occupies 4, `%5.2f` 6 (a field that grows because the value does not fit), and `%6s` 6 characters.

Width is most useful when printing tables. When the same specifiers are used on every row, the columns line up neatly one under the other. Text columns are usually left-aligned (`%-10s`) and number columns right-aligned (`%6d`, `%8.2f`). That way the units digits of the numbers fall one under another.

Example 3.4.3

Let us find, character by character, the output of the following line with the US English locale:

```
String name = "Ali";
double grade1 = 72.456;
System.out.printf("[%-5s][%3d][%7.2f]%n", name, 7, grade1);
```

`%-5s` : `"Ali"` has three letters, is left-aligned in a field of five characters and gets two spaces on its right: `Ali··`. `%3d` : 7 has one digit and gets two spaces on its left: `··7`. `%7.2f` : 72.456 is rounded to two decimals and becomes 72.46. This text is 5 characters; to fill the field of seven characters, two spaces are added at the start: `··72.46`. The result:

```
[Ali  ][  7][  72.46]
```

The rounding rule is also worth knowing. `printf` looks at the decimal representation of the number and rounds halves away from zero (`HALF_UP`). Unlike `Math.round`, it also moves away from zero for negative halves:

```
System.out.printf("[%.2f][%.2f][%.0f][%.0f][%.0f]%n", 2.345, 2.355, 0.5, 1.5, -2.5);
System.out.printf(Locale.US, "%,.2f %08.3f %+d %X%n", 1234567.891, 3.14159, 5, 255);
```

```
[2,35][2,36][1][2][-3]
1,234,567.89 0003.142 +5 FF
```

2.345 and 2.355 are rounded up as expected. 0.5 → 1, 1.5 → 2 and −2.5 → −3. The flags in the second line are explained in the next subsection.

3.4.6.3 Other formats and String.format

```
System.out.printf("|%5s|%-5s|%05d|%.1f|%n", "ab", "ab", 42, 3.14159);
System.out.printf("%c %b %e %x %%%n", 'Z', true, 1234.5, 255);
String msg = String.format("%s: %d points", "Ayşe", 85);
System.out.println(msg);
System.out.printf("%,d%n", 1234567);
System.out.printf(Locale.US, "%.2f%n", 2.5);       // import java.util.Locale;
System.out.println("%s = %d".formatted("x", 7));   // Java 15+
```

```
|   ab|ab   |00042|3,1|
Z true 1,234500e+03 ff %
Ayşe: 85 points
1.234.567
2.50
x = 7
```

The new elements in this example are the following:

- `%05d`: the `0` flag fills the field with zeros instead of spaces: `00042`. It is used for fixed-length numbers such as invoice and student numbers.
- `%e` is scientific notation: 1234.5 = 1.2345 × 10^3, in the output `1,234500e+03`. `%x` writes in hexadecimal: 255 → `ff`; `%X` for uppercase. `%%` writes the percent sign itself.
- `String.format` uses the same rules but does not write to the screen; it returns the formatted text as a `String`. The result can be stored in a variable, written to a file or appended to another text.
- `%,d` adds a thousands separator. With the Turkish locale the separator is a point: `1.234.567`. With the US locale it is a comma.
- The first argument of `printf` can be a `Locale`. When `Locale.US` is given, the decimal separator becomes a point: `2.50`. Data that another program will read (such as a CSV file) is written this way.
- The `formatted` method that came with Java 15 is called on the format string: `"%s = %d".formatted("x", 7)` and `String.format("%s = %d", "x", 7)` give the same result.

For the compiler the format string is ordinary text. If a specifier and the type of its argument do not match, the error appears not at compile time but at run time. The call `printf("%d", 3.5)` gets a `double` instead of the integer that `%d` expects and throws an exception:

```
Exception in thread "main" java.util.IllegalFormatConversionException:
    d != java.lang.Double
```

The message means “the specifier `d` cannot be used with `Double`”. `printf` does not convert; it does not cut the number and write 3. The right way is to use `%f` or `%.0f`. If there are more specifiers than arguments, `MissingFormatArgumentException` is thrown. `%s`, on the other hand, works with every type, because every value can be converted to text.

3.4.7 Applications

When the tools of the earlier sections, that is, indexing, loops, comparison and concatenation, are used together, real problems get solved. This section treats in detail the two applications in the slides and the small methods in the exercise slides.

3.4.7.1 Palindrome check

Definition 3.4.13

A **palindrome** is a text that reads the same backward as forward: “kayak”, “level”, “noon”. In sentences, spaces and punctuation are ignored.

The most direct way to tell whether a text is a palindrome is to reverse the text and compare it with the original:

```
String text = "Kayak";
text = text.toLowerCase();
String rev = "";
for (int i = text.length() - 1; i >= 0; i--) {
  rev += text.charAt(i);
}
System.out.println(rev);
System.out.println(text.equals(rev));
```

```
kayak
true
```

The code works in three steps. First the text is converted to lowercase; if it were not, `K` and `k` would count as different. Because `toLowerCase()` returns a new string, the result is assigned to `text` again. In the second step an empty `rev` string is prepared and the loop goes down from the last index to 0. On each pass the character at that index is appended to the end of `rev`. In the third step the two strings are compared with `equals`. Table 3.4.12 gives the passes of the loop.

`i`	`i >= 0`	`charAt(i)`	`rev` **(at the end of the pass)**
4	true	`'k'`	`"k"`
3	true	`'a'`	`"ka"`
2	true	`'y'`	`"kay"`
1	true	`'a'`	`"kaya"`
0	true	`'k'`	`"kayak"`
−1	false	—	the loop ends

Table 3.4.12: The reversing loop for `text = "kayak"`.



Figure 3.4.14: `text` and `rev` when the loop ends. The contents are the same; `equals` returns `true`.

The comparison must be done with `equals`. `rev` is a new object built by concatenation at run time; `text == rev` would always give `false`. The line `rev += …` in the loop produces a new string on every pass. For short texts this does not matter, but the same job is done both briefly and efficiently with `new StringBuilder(text).reverse().toString()`.

For sentences, the characters that are not letters are first deleted. The call `text.replaceAll("[^\\p{L}]", "")` uses a regular expression: `\p{L}` stands for every letter in Unicode and `[^…]` for "every character other than these". Only the letters remain. When the Turkish sentence "Ey Edip Adana'da pide ye" is cleaned and lowercased, it becomes `"eyedipadanadapideye"`, and this is a palindrome.

The reversing method creates a copy of the text. The same check can be done without using memory, with two indices: one starts from the front, the other from the end, and they compare characters until they meet in the middle. As soon as an inequality is found the result is `false`; there is no need to look at the remaining characters. The cost of both methods is proportional to the length of the text.

3.4.7.2 Word frequency

Counting how many times each word occurs in a text is one of the most basic problems of text processing. The solution in the slides works only with arrays and two nested loops:

```
String text = "One two one six TWO one";
String[] words = text.toLowerCase().split(" ");
boolean[] counted = new boolean[words.length];
for (int i = 0; i < words.length; i++) {
  if (counted[i]) {
    continue; // this word was already counted
  }
  int freq = 1;
  for (int j = i + 1; j < words.length; j++) {
    if (words[i].equals(words[j])) {
      freq++;
      counted[j] = true;
    }
  }
  System.out.println(words[i] + ": " + freq);
}
```

```
one: 3
two: 2
six: 1
```

The code follows these steps:

- `text.toLowerCase().split(" ")` first lowercases the text and then splits it at the spaces. Thanks to the lowercasing, “One/one” and “TWO/two” count as the same word.
- The `counted` array keeps a flag for each word. When a `boolean` array is created, all the elements are `false`.
- The outer loop looks at each word once. If the flag of the word is `true`, that word has been counted before; `continue` skips the rest of the pass.
- The inner loop scans only the words to the right. For each matching word the counter increases and the flag of the match is set to `true`. This way the same word is not reported a second time.

The words are compared with `equals`. The pieces that `split` produces are separate objects created at run time. Had `==` been used, no match would have been found and the frequency of every word would have come out as 1. This was verified by running: with `==` the word “one” is counted only once.

The trace slide in the slides runs the same code on the text `"apple Pear apple plum PEAR"`:

```
String text = "apple Pear apple plum PEAR";
String[] k = text.toLowerCase().split(" ");
boolean[] counted = new boolean[k.length];
for (int i = 0; i < k.length; i++) {
  if (counted[i]) continue;
  int freq = 1;
  for (int j = i + 1; j < k.length; j++) {
    if (k[i].equals(k[j])) {
      freq++;
      counted[j] = true;
    }
  }
  System.out.println(k[i] + ": " + freq);
}
```

```
apple: 2
pear: 2
plum: 1
```

`i`	`k[i]`	**Matching `j`**	`freq`	`counted`	**Output**
0	egg	2	2	F F **T** F F	`apple: 2`
1	apple	4	2	F F T F **T**	`pear: 2`
2	egg	—	—	F F T F T	flag set, `continue`
3	plum	none	1	F F T F T	`plum: 1`
4	apple	—	—	F F T F T	flag set, `continue`

Table 3.4.13: Trace of the word frequency. `k = ["apple", "pear", "apple", "plum", "pear"]`; the flags are at the end of each outer pass.



Figure 3.4.15: The word array and the flags at the end of the pass `i` = 0. The “apple” at index 2 has been marked as counted.

The cost of the algorithm grows with the square of the number of words. For *n* words the inner loop makes *(n − 1) + (n − 2) + … + 1 = n(n − 1)/2* comparisons in the worst case. For a text of a few sentences this is very little. If all the words of a book are to be counted, a collection that keeps key-value pairs (`HashMap`) gives a much faster result.

3.4.7.3 Small string methods

The exercise and example-problem slides of the presentation contain a few short methods that combine string methods. These methods show common patterns of indexing and looping.

Reversing. The `reverse` method moves the loop in the palindrome example into a method:

```
static String reverse(String s) {
  String rev = "";
  for (int i = s.length() - 1; i >= 0; i--) {
    rev += s.charAt(i);
  }
  return rev;
}
```

The call `reverse("Giresun")` returns `nuseriG`. The capital `G` has moved to the end; the method does not touch letter case.

Counting characters. The `count` method finds how many times a character occurs:

```
static int count(String s, char target) {
  int count = 0;
  for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == target) {
      count++;
    }
  }
  return count;
}
```

The comparison `s.charAt(i) == target` is correct, because both sides are `char` and `char` is a primitive type. Here `==` compares values, not references. `count("Karadeniz", 'a')` returns 2: `K` is an uppercase letter and does not match `'a'`; the only matches are the `a`s at indices 1 and 3.

Capitalizing the first letter. `capitalize` makes the first letter uppercase and the rest lowercase:

```
static String capitalize(String word) {
  return word.substring(0, 1).toUpperCase()
       + word.substring(1).toLowerCase();
}
```

The method handles the two parts separately and joins them: `substring(0, 1)` is the first letter and `substring(1)` is the rest. `capitalize("jAVA")` → `Java`. With the Turkish locale, `capitalize("istanbul")` → `İstanbul` and `capitalize("ılgaz")` → `Ilgaz`. If it is called with the

empty string, `substring(0, 1)` asks for an invalid range and a `StringIndexOutOfBoundsException` is thrown. A robust version first checks `if (word.isEmpty()) return word;`.

Counting vowels. `indexOf` can be used to tell whether a character is in a set:

```
String word = "Ordu";
String vowels = "aeıioöuüAEIİOÖUÜ";
int count = 0;
for (int i = 0; i < word.length(); i++) {
  char c = word.charAt(i);
  if (vowels.indexOf(c) != -1) {
    count++;
  }
}
System.out.println("Vowels: " + count);
```

```
Vowels: 2
```

The string `vowels` contains the eight vowels of Turkish in their uppercase and lowercase forms. For each character `vowels.indexOf(c)` is called; if the result is not −1, the character is a vowel. In `"Ordu"`, `O` and `u` are vowels: `Vowels: 2`. This pattern is both shorter and more readable than an `if` chain with eight branches.

Number of words. If there is a single space between the words, the number of words is found in one line: `sentence.split(" ").length`. For `"Java and object oriented programming"` the result is 5. If there can be several spaces between the words, `sentence.trim().split(" +").length` is written.

Example 3.4.4

The first example problem in the slides tests immutability, `substring`, `indexOf` and the order of concatenation together:

```
String s = "Karadeniz";
s.toUpperCase();
String t = s.substring(2, 5);
System.out.println(t + " " + t.length());
System.out.println(s.indexOf("de"));
System.out.println(s.indexOf("x"));
System.out.println(1 + 2 + "3" + 4 + 5);
```

```
rad 3
4
-1
3345
```

Because the result of `s.toUpperCase()` is not assigned, `s` does not change. The indices of `"Karadeniz"` are K0 a1 r2 a3 d4 e5 n6 i7 z8. `substring(2, 5)` takes indices 2, 3, 4: `"rad"`, length 3. `"de"` starts at index 4. `"x"` is not in the string, −1. The last line is computed from left to right: 1 + 2 = 3 (two numbers), 3 + `"3"` = `"33"`, then `"334"` and `"3345"`.

The second problem tests the pool:

```
String a = "java";
String b = "ja" + "va";
String c = new String("java");
String d = c;
System.out.println(a == b);
System.out.println(a == c);
System.out.println(c == d);
System.out.println(a.equals(c));
```

```
true
false
true
true
```

`"ja" + "va"` is a constant computed at compile time and points to the same object as `a` in the pool: `a == b` is `true`. `c` is a separate object created with `new`: `a == c` is `false`. `d = c` copies the reference; both are the same object: `true`. Because the contents are the same, `a.equals(c)` is `true`.

The third problem extracts initials and parts of an e-mail address:

```
public static String initials(String name, String surname) {
  return name.charAt(0) + "." + surname.charAt(0) + ".";
}
System.out.println(initials("Sercan", "Külcü"));
String e = "student@giresun.edu.tr";
int at = e.indexOf('@');
System.out.println(e.substring(0, at));
System.out.println(e.substring(at + 1));
```

```
S.K.
student
giresun.edu.tr
```

In the expression `name.charAt(0) + "."`, the right side is a string, so concatenation is done: `"S."`. If two `char`s had been added directly, the result would have been a number; `'S' + '.'` gives 83 + 46 = 129. The e-mail part is the same as the example solved earlier.

3.4.8 Common mistakes

The compiler catches some of the mistakes with strings; if its message is read, they are easy to fix. Others compile and then give wrong results or throw exceptions when run. Table 3.4.14 summarizes the first group and Table 3.4.15 the second. The messages were taken from the `javac` of JDK 21.

Mistake	Compiler message	Fix
`String s = 'Hello';`	`unclosed character literal`	Single quotes are only for a single character; a string is written with double quotes.
`char c = "a";`	`incompatible types: String cannot be converted to char`	`char c = 'a';` or `"a".charAt(0)`.
`s.length`	`cannot find symbol` / `symbol: variable length`	In a string `length()` is a method. For an array it is the opposite: `d.length`.
`string s = …;`	`cannot find symbol` / `symbol: class string`	The class name is `String`; Java is case-sensitive.
`int n = "42";`	`incompatible types: String cannot be converted to int`	`Integer.parseInt("42")`.
`s.charAt(0) = 'j';`	`unexpected type` / `required: variable`	A string is immutable; use a new string or `StringBuilder.setCharAt`.
`s[0]`	`array required, but String found`	`s.charAt(0)`.
`if (a < b)` (strings)	`bad operand types for binary operator '<'`	`a.compareTo(b) < 0`.

Table 3.4.14: String mistakes caught by the compiler.

For the error in the first line `javac` actually gives three messages: `unclosed character literal` twice and `not a statement` once. The compiler takes the single quote to be the start of a character literal, sees that the quote is not closed, and cannot make sense of what follows. In such chained messages the first message shows the real cause:

```
Quote.java:3: error: unclosed character literal
        String s = 'Hello';
                   ^
```

Mistake	Symptom	Fix
Comparing the content with `==`	The condition is `false` even though the user input is right.	`"yes".equals(input)`
Forgetting to assign the result	`s` does not change after `s.toUpperCase();`.	`s = s.toUpperCase();`
Thinking the last index is `length()`	`StringIndexOutOfBoundsException: Index 4 out of bounds for length 4`	`s.charAt(s.length() - 1)`; `i < s.length()` in loops
`nextLine` after `nextInt`	The line is read as empty and the question is skipped.	Put a `keyboard.nextLine();` in between
Calling a method on a `null` string	`NullPointerException: Cannot invoke "String.equals(Object)"`	Put the constant first: `"yes".equals(answer)`
Thinking the end of `substring` is included	`s.substring(5, 12)` gives 7 characters, not 8.	Length = end − start.
Addition without parentheses	`"Total: " + x + y` → `Total: 105`	`"Total: " + (x + y)`
Case conversion with the default locale	`"TITLE".toLowerCase()` gives `tıtle` with the Turkish locale.	`toLowerCase(Locale.ROOT)` or `equalsIgnoreCase`
A wrong specifier in `printf`	`IllegalFormatConversionException`	`%d` for integers, `%f` for floating-point numbers.

Table 3.4.15: String code that compiles but runs wrongly.

What the mistakes in the second table have in common is that the compiler cannot see them. `==` and `equals` are both valid Java. A method call whose result is not assigned is a valid statement too. The best protection against these mistakes is to try the program with different inputs and compare the output with the expected value. The empty string, a one-character string, Turkish letters and leading and trailing spaces in particular make good test inputs.

3.4.9 Summary

- **String** is an immutable object that stores a sequence of characters. It is created with a string literal (`"…"`) or with `new`; since it is in the `java.lang` package it needs no `import`.
- **Concatenation:** `+` is evaluated from left to right. If one of the operands is a string, concatenation is done; if both are numbers, addition. `char` is a number: `'a' + 'b'` = 195. Parentheses are used to add numbers.
- The **index** runs from 0 to `length() - 1`. `charAt(i)` returns a `char`. An index that goes outside throws `StringIndexOutOfBoundsException`. In a string `length()` is a method, in an array `length` is a field.
- **Methods:** `indexOf` (−1 if not found), `substring(a, b)` (`b` excluded, length `b - a`), `toUpperCase`, `replace`, `trim`, `split`, `join`; all return a new value and the result must be assigned.

- **Locale:** case conversion without a parameter depends on the setting of the machine. With the Turkish locale `i ⇔ İ`, `ı ⇔ I`. `Locale.ROOT` is used for texts inside the program.
- **StringBuilder:** a mutable buffer. Concatenation with `+=` in a loop does $O(n^2)$ work, with `append` $O(n)$. When the capacity is full it roughly doubles (20 → 42).
- **Scanner:** `next()` reads a word and `nextLine()` the rest of the line. After `nextInt()` the ⏎ stays in the buffer; a `nextLine()` that follows returns the empty string.
- `==` / `equals`: `==` asks whether it is the same object, `equals` asks whether the content is the same. The content of strings is always compared with `equals`; against `null` the constant is put first.
- **String pool:** string literals with the same content share one object. `new` and run-time concatenation produce a new object; `intern()` gives the pool counterpart.
- **compareTo:** returns the code difference of the first differing character, or the length difference in the case of a prefix. It uses Unicode order; for Turkish alphabet order `Collator` is needed.
- **printf:** `%d`, `%f`, `%s`, `%c`, `%b`, `%n`. `%Wd` right-aligns and `%-Wd` left-aligns; `%.2f` writes two decimal digits. A value that does not fit enlarges the field. Separators depend on the locale.

3.4.10 Key terms

string string literal string concatenation character (char) Unicode index indexing immutable reference heap garbage collector locale buffer string pool null reference format specifier compile time / run time runtime error exception literal reference type

---

3.4.11 Test Yourself

1. What does `System.out.println(2 + 3 + "x" + 2 + 3);` print? How is the expression changed so that the sum of the last two numbers is printed?

   Answer

   It prints `5x23`. The two numbers on the left are added (5) before a string is met. Every `+` after `"x"` is concatenation. For the sum at the end, parentheses are needed: `2 + 3 + "x" + (2 + 3)` → `5x5`.

2. For `String s = "Karadeniz Teknik";`, what are `s.indexOf(' ')`, `s.substring(s.indexOf(' ')`

`+ 1)` and `s.lastIndexOf('e')`?

Answer

"Karadeniz" has nine letters; the space is at index 9. `substring(10)` gives what follows the space: `Teknik`. `T` is at index 10, `e` at index 11, and there is no other `e` after it: `lastIndexOf('e')` = 11.

3. After the lines `StringBuilder sb = new StringBuilder("abc"); sb.append("de").insert(2, "X").reverse();`, what are the content and the length of `sb`? How many objects would be created if the same operations were done with `String`?

Answer

`"abc"` → `append` → `"abcde"` → `insert(2, "X")` → `"abXcde"` → `reverse` → `edcXba`, length 6. All the operations are done on the same object. With `String` each step would produce a new string: three new objects besides the initial string.

4. What do `"Ankara".compareTo("Antalya")` and `"abc".compareTo("ab")` return? If these strings are put in alphabetical order, which comes first?

Answer

In the first pair the strings differ at index 2, `'k'` (107) versus `'t'` (116): 107 − 116 = −9. Because it is negative, "Ankara" comes first. In the second pair `"ab"` is a prefix of `"abc"`; the result is the difference of the lengths: 3 − 2 = 1. "ab" comes first.

5. After the lines `String x = "Ada"; String y = "A" + "da"; String z = new String("Ada"); String w = z.intern();`, what do `x == y`, `x == z`, `x == w` and `z.equals(y)` give?

Answer

`true false true true`. `"A" + "da"` becomes the constant `"Ada"` at compile time and points to the object in the pool. `z` is a separate object outside the pool. `intern()` returns its counterpart in the pool, that is, the object of `x`. Because the contents are the same, `equals` gives `true`.

6. What is the output of `System.out.printf("[%6.1f][%-4s][%3d]%n", 9.96, "ok", 12345);` on a

computer set to Turkish?

Answer

`[ 10,0][ok ][12345]`. When 9.96 is rounded to one decimal it becomes 10.0; this is 4 characters and two spaces are added at the start. `"ok"` is left-aligned and two spaces come to its right. 12345 has five digits and does not fit in a width of 3; the field grows and the number is not cut.

7. At the keyboard, first `␣␣42␣␣Ayşe Yılmaz⏎` and then `Son⏎` are typed. After the lines `int n = k.nextInt(); String r = k.nextLine(); String t = k.next();`, what are `n`, `r` and `t`?

Answer

`nextInt()` skips the leading spaces and reads 42. `nextLine()` does not skip spaces; it takes the rest of the line: `r = "  Ayşe Yılmaz"` (two spaces at the start). Because the ⏎ has been consumed, `next()` reads the word on the second line: `t = "Son"`.

8. What does `String u = "Banana"; u.replace('a', 'e'); u.concat("!"); System.out.println(u);` print? How is the code fixed if the programmer wants to see `Benene!`?

Answer

It prints `Banana`. Both methods return a new string, but the results are lost because they are not assigned. The fixed form: `u = u.replace('a', 'e').concat("!");` or two separate assignments.

9. How many elements does the array returned by `"a-b--c".split("-")` have? What does `"KIRMIZI".toLowerCase().equals("kirmizi")` give on a computer set to Turkish?

Answer

An empty piece is produced between two consecutive hyphens: `["a", "b", "", "c"]`, 4 elements. The second expression gives `false`: when `I` is lowercased with Turkish rules it becomes `ı` and the result is `"kırmızı"`. `toLowerCase(Locale.ROOT)` or `equalsIgnoreCase` gives `true`.

3.4.12 Further reading

- Oracle, *The Java Tutorials*, the section “Numbers and Strings”: the official tutorial that explains `String`, `StringBuilder` and formatted number output with short examples.
- Oracle, *Java SE 21 API Documentation*, the class `java.util.Formatter`: the complete list of all the flags, the width and precision rules and the type letters that can be used in `printf`.

- J. Gosling et al., *The Java Language Specification*, Java SE 21, §3.10.5 and §15.18.1: define precisely the sharing of string literals in the pool and the rules for concatenation with `+`.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, Chapter 3.6: treats strings, `StringBuilder` and the Java counterpart of Unicode from the point of view of an experienced programmer.

3.4.13 References

1. Oracle, *The Java Tutorials*: “Strings”, “Comparing Strings and Portions of Strings”, “The StringBuilder Class”, “Formatting Numeric Print Output”.
2. Oracle, *Java SE 21 API Documentation*: `java.lang.String`, `java.lang.StringBuilder`, `java.util.Scanner`, `java.util.Formatter`, `java.util.Locale`.
3. J. Gosling vd., *The Java Language Specification*, Java SE 21 Edition (§3.10.5 String Literals, §15.18.1 String Concatenation Operator +).
4. P. Deitel, H. Deitel, *Java: How to Program*, 11th ed., Pearson, 2017 (Chapter 14: Strings, Characters and Regular Expressions).
5. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., 2022 (Chapter 3.6: Strings).
6. A. B. Downey, C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019 (Chapter 6: Loops and Strings).

PART I · CHAPTER 3.5

Recursion

Some problems contain a smaller copy of themselves. 5! is 5 times 4!; the digit sum of 48729 is the digit sum of 4872 plus 9. Recursion turns this structure directly into code: a method hands a smaller piece of the work back to itself and builds its own answer from the reply. The chapter first introduces the base case, the recursive step and the call stack. It then writes factorial, sum, four number sequences, digit sum, palindrome testing, Euclid's algorithm and the Towers of Hanoi in Java and traces each call by call. The last sections deal with the price of recursion: stack overflow, recomputing the same value over and over, conversion to loops, tail recursion and memoization. Every program was run on JDK 21; the outputs, call counts and timings come from those runs.

LEARNING OUTCOMES

Concept
Define recursion, the base case and the recursive step; reduce a problem to a smaller copy of itself.

How it runs
Trace, step by step, how frames pile up on the call stack and unwind again; draw a call tree.

Writing
Write recursive Java methods for factorial, sum, number sequences, digit sum, palindromes, the GCD and the Towers of Hanoi.

Pitfalls
Recognize and prevent infinite recursion, `StackOverflowError` and missing `return` statements.

Efficiency
Compare recursive and iterative solutions; convert a recursive method into a loop when needed.

Advanced techniques
Explain tail recursion and memoization; eliminate repeated computation.

3.5.1 The idea of recursion

A method names a piece of work and makes it callable in one line. `main` can call a method, and that method can call another. The method it calls may even be itself. Java does not forbid this, and the possibility lets some problems be written far more briefly and clearly than with loops.

3.5.1.1 What is recursion?

> **Definition 3.5.1**
>
> **Recursion** is a method calling itself in order to split its task into smaller parts. Each call solves a smaller but otherwise identical copy of the problem; the smallest copy is solved directly. A method that calls itself is called a **recursive method**.

Java has no special keyword for recursion. The method body simply calls its own name, just like any other method call. To the compiler, `factorial` calling `factorial` is no different from `main` calling `println`. The difference shows up at run time: several calls of the same method wait in memory at once.

Recursion has three familiar counterparts in everyday life. In a set of **matryoshka dolls** every doll contains a smaller doll. The smallest doll is not opened; that is the base case of recursion. In a **cinema**, someone sitting in the dark wants to know which row they are in. They ask the person in front, who asks the person in front of them, and so on. The person in the first row says "1". As the answers come back, each person adds 1 to the number they hear and tells the person behind. The question travels forward and the answer travels back. This is exactly how a recursive method runs. Between **two facing mirrors**, an image contains an image, which contains yet another image, and the sequence never ends. That is a picture of recursion without a base case.

Recursion is not the right choice for every problem. For some kinds of problems, however, it gives a much clearer solution than a loop. Tree-shaped data, such as folders and subfolders, is one of them. Number sequences defined in terms of themselves are another. Divide-and-conquer algorithms, which split a problem in two and solve the parts separately, belong to the same group. Some puzzles, such as the Towers of Hanoi, are hard to write with loops but take only a few lines with recursion.

3.5.1.2 Base case and recursive step

Every recursive method has two parts. One makes it stop, the other makes it progress.

Definition 3.5.2

The **base case** is the smallest problem, whose answer is known directly. In this case the method does not call itself and the recursion stops. The **recursive step** reduces the problem by one step, asks the method itself, and builds its own answer from the reply.

Factorial shows these two parts most clearly. 1! = 1 is known; no computation is needed. 4! is not known directly, but if 3! is known, then 4! = 4 × 3!. The inductive definition from mathematics corresponds one-to-one to the recursive method:

> $n! = 1$ (if $n \le 1$), $n! = n \times (n - 1)!$ (if $n > 1$)

The first line is the base case and the second is the recursive step. The definition may look circular, but it is not: defining 4! uses 3!, and defining 3! uses 2!. At every step the argument shrinks, and in the end it reaches 1. Figure 3.5.1 shows this nesting with boxes.

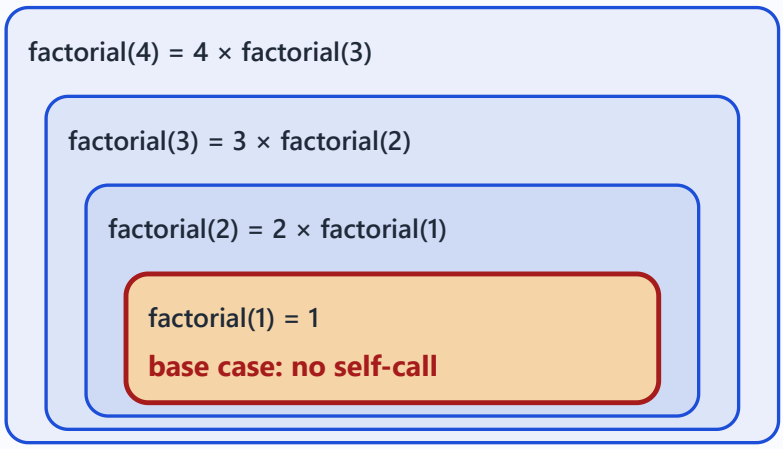


Figure 3.5.1: The problem `factorial(4)` shrinks by one at each level until it reaches the base case `factorial(1)`. The innermost box does not call itself; its answer is simply 1.

Both parts are mandatory. Without a base case the method calls itself forever. If the recursive step does not shrink the problem, the base case is never reached and the result is the same. The two basic rules every recursive method must obey follow from these two conditions.

3.5.1.3 The call stack

To understand how recursion works, one must know how Java keeps method calls in memory.

Definition 3.5.3

For every method call, a **stack frame** is opened in memory. The frame holds the call's parameters, its local variables and the place to return to when the method finishes. Frames are piled on top of each other on the **call stack**; only the top frame runs, and the ones below wait.

A stack is like a pile of plates. A new plate goes on top, and the next plate to be taken also comes from the top. This discipline is called **last in, first out** (LIFO). When a method is called, its frame is added to the top of the stack. When the method finishes with `return`, its frame is removed and the value it returns is passed to the frame below. That frame resumes exactly where it left off.

In recursion, several frames of the same method live at the same time. When `factorial(4)` calls `factorial(3)`, the first frame is not removed; it waits for the value coming up from below so that it can do its multiplication. Every frame has its own `n`. The stack can hold four separate `n`s: one is 4, one is 3, one is 2, one is 1. They are variables that share a name but are independent of each other. Changing `n` in one frame does not affect the others.



Figure 3.5.2: The deepest moment of the call `factorial(4)`. `factorial(1)` runs at the top of the stack; the three frames below it wait to complete their products, and at the very bottom `main` waits for the result. A frame is added on a call (left arrow) and removed on a return (right arrow).

Figure 3.5.2 shows the deepest moment of a program in which `main` calls `factorial(4)`. At that moment the stack holds five frames: `main` and four `factorial` frames. Only `factorial(1)` at the top is running. Each frame below is stopped halfway through a multiplication, waiting for the missing factor: `4 * ?`, `3 * ?`, `2 * ?`.

The size of the stack is limited. In Java, each thread's stack is a fixed region of memory set when the program starts. A very deep recursion fills this region and the program stops with a `StackOverflowError`. This limit is the most important practical constraint of recursion.

> **Common mistake**
>
> Believing that the variable is "overwritten" in a recursive call. When `factorial(3)` is called, the `n` of `factorial(4)` does not become 3. A new `n` is created in the new frame, and the old frame's `n = 4` waits unchanged. That is why the multiplication on the way back can be carried out as `4 * 6`.

3.5.2 Factorial and sum

Factorial and the sum from 1 to n are the two simplest examples of recursion. Both follow the same pattern: reduce the argument by one, call yourself, and add your own contribution to the result that comes back. Once this pattern is understood well, most of the later examples read as variations on it.

3.5.2.1 The factorial method

The factorial of n is the product of the numbers from 1 to n: n! = 1 × 2 × … × n. By definition 0! = 1. The same product can also be written recursively as n! = n × (n − 1)!. In Java it looks like this:

```
int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;   // base case
    } else {
        return n * factorial(n - 1); // recursion
    }
}
```

Line 1 is the method header: it takes an `int` parameter and returns an `int` value. Lines 2 and 3 are the base case. If `n` is 0 or 1, the answer is 1 and the method returns without calling itself. Line 5 is the recursive step: n! is n times (n − 1)!. The multiplication cannot be done before `factorial(n - 1)` has been evaluated, so the method first calls itself, waits for the answer, then multiplies and returns.

One can show that the method always terminates. On every call `n` decreases by 1. If `n ≥ 0` at the start, this decrease eventually reaches 1 or 0 and the base case runs. This guarantee of progress is the property to look for in every recursive method. The call `factorial(5)` returns 120 and `factorial(0)` returns 1.

n	`factorial(n)`	True n!	Correct?
0	1	1	yes

n	`factorial(n)`	True n!	Correct?
5	120	120	yes
12	479001600	479,001,600	yes
13	1932053504	6,227,020,800	no (overflow)
17	−288522240	355,687,428,096,000	no (even negative)

Table 3.5.1: Some values of `factorial` returning `int` (JDK 21).

That the code is mathematically correct does not mean every result will be correct. An `int` can hold at most 2,147,483,647. 12! = 479,001,600 is below this limit, while 13! = 6,227,020,800 is above it. Java does not report overflow: the high bits of the product that do not fit are discarded, and a meaningless number is left (Table 3.5.1). `factorial(13)` returns 1932053504; for 17! the result is negative. If the return type is changed to `long`, results are correct up to 20! = 2,432,902,008,176,640,000; 21! does not fit in a `long` either. For larger factorials, `java.math.BigInteger` is used.

Common mistake

Expecting the compiler or the runtime to catch overflow. In `int` and `long` multiplication, Java produces no warning and no exception. The range of the result must be estimated in advance and a suitable type chosen. If overflow should be reported as an exception, use `Math.multiplyExact`; on overflow this method throws an `ArithmeticException`.

3.5.2.2 Expanding factorial(4)

The shortest way to find the result of a recursive call by hand is to expand it by substitution. Wherever `factorial(k)` appears, put `k * factorial(k - 1)`, and when the base case is reached, compute back:

```
factorial(4)
  = 4 * factorial(3)
  = 4 * (3 * factorial(2))
  = 4 * (3 * (2 * factorial(1)))
  = 4 * (3 * (2 * 1))
  = 4 * (3 * 2)
  = 4 * 6
  = 24
```

The expansion has three phases. The first four lines are the **descent**. Every call opens a new call and the multiplications are put on hold, because the value on the right is not yet known. In the fifth line the **bottom**, that is, the base case, is reached. `factorial(1)` returns 1 directly. Because the code stops at `n == 1`, `factorial(0)` is never called. The last three lines are the **ascent**. The pending multiplications are completed from the inside out: first 2 × 1 = 2, then 3 × 2 = 6, and finally 4 × 6 = 24.

The nesting of the parentheses is the call stack written out in text. The innermost parenthesis is the top of the stack and is completed first. Each closing parenthesis corresponds to a frame removed from the stack. Note that the multiplications are performed in reverse order, not in the order 4 × 3 × 2 × 1. For multiplication the order does not change the result; but for work where order matters, such as printing, the difference between the descent and the ascent is decisive.

3.5.2.3 Tracing on the call stack

The same computation can also be traced as the run of a complete program. In the program below, `main` calls `factorial(4)` and prints the result:

```
static int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

public static void main(String[] args) {
    System.out.println(factorial(4));
}
```

```
24
```

Table 3.5.2 gives the call stack at each step of the program, from bottom to top. The "Pending work" column is the computation the frame will carry out when it resumes.

Step	Event	Stack (bottom to top)	Pending work
1	`main` calls `factorial(4)`	main	print the result
2	`n = 4` : condition false, calls `factorial(3)`	main, f(4)	`4 * ?`
3	`n = 3` : condition false, calls `factorial(2)`	main, f(4), f(3)	`3 * ?`
4	`n = 2` : condition false, calls `factorial(1)`	main, f(4), f(3), f(2)	`2 * ?`
5	`n = 1` : base case, returns 1	main, f(4), f(3), f(2), f(1)	—
6	`f(2)` receives 1: returns `2 * 1 = 2`	main, f(4), f(3), f(2)	—
7	`f(3)` receives 2: returns `3 * 2 = 6`	main, f(4), f(3)	—
8	`f(4)` receives 6: returns `4 * 6 = 24`	main, f(4)	—
9	`main` prints 24	main	—

Table 3.5.2: Step-by-step trace of the call `factorial(4)` .

Four calls of `factorial` are made. At the deepest moment, step 5, the stack holds five frames: `main` and four `factorial` frames. Figure 3.5.3 shows three moments of the same trace as a call chain. The top shows the bottom of the descent and the bottom shows the state after all the returns have completed.

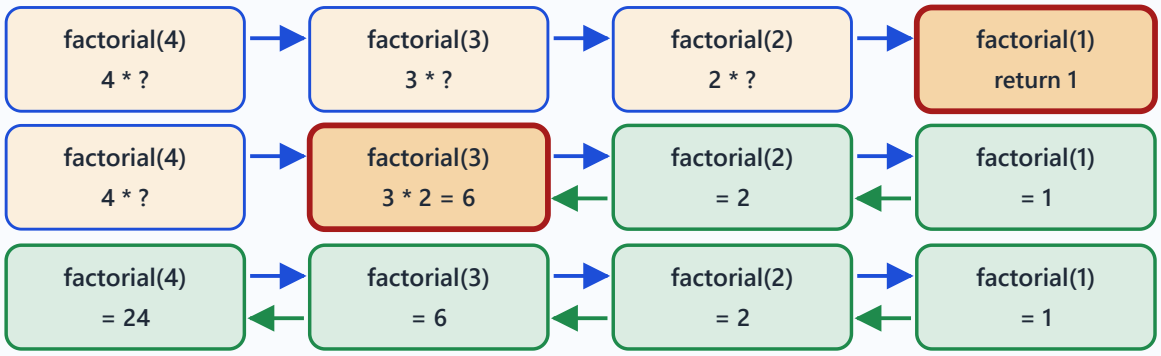


Figure 3.5.3: The call chain of `factorial(4)`. Top: the base case has been reached; three frames wait for the products `4 * ?`, `3 * ?` and `2 * ?`. Middle: `factorial(2)` has returned 2, and `factorial(3)` now computes `3 * 2 = 6`. Bottom: all frames have returned a value; `factorial(4) = 24`.

The time and memory cost of a recursive method can be read off this trace. `factorial(n)` makes exactly n calls, and each call has one comparison and one multiplication. The running time is proportional to n, that is, O(n). At most n frames are on the stack at once, so the memory is also O(n). A loop doing the same job also takes O(n) time, but it runs in a single frame and needs only O(1) memory. As the depth grows, this difference becomes important.

3.5.2.4 The sum from 1 to n

The sum of the numbers from 1 to n is written with the same pattern: sum(n) = n + sum(n − 1) and sum(1) = 1. The only differences from factorial are that addition replaces multiplication and that the base case is just `n == 1`.

```
int sum(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n + sum(n - 1);
    }
}
```

```
sum(5) = 5 + sum(4)
          = 5 + (4 + (3 + (2 + sum(1))))
          = 5 + (4 + (3 + (2 + 1)))
          = 15
```

In the expansion, each call leaves behind an addition that is waiting for its number. When the bottom is reached, 2 + 1 = 3 is computed, then 3 + 3 = 6, 4 + 6 = 10 and 5 + 10 = 15. In the same way `sum(100)` gives 5050. This result is confirmed by Gauss's formula n(n + 1)/2: 100 · 101 / 2 = 5050.

This method has a weak point. The base case is only `n == 1`. When it is called with `sum(0)` or a negative number, `n` never becomes 1: it decreases forever as 0, −1, −2, ... The result is infinite recursion, and the program stops with a `StackOverflowError`. In the safe version the base case is `if (n <= 0) return 0;`. This version also catches invalid inputs, and for `sum(1)` it correctly gives 1 + 0 = 1.

Example 3.5.1

The `sum` method with the safe base case expands as follows for `sum(3)`:

1. `sum(3)`: 3 ≤ 0 is false; `3 + sum(2)` is pending.
2. `sum(2)`: `2 + sum(1)` is pending.
3. `sum(1)`: `1 + sum(0)` is pending.
4. `sum(0)`: 0 ≤ 0 is true; base case, returns 0.
5. On the way back, 1 + 0 = 1, 2 + 1 = 3 and 3 + 3 = 6 are computed. The result is 6.

This is one call more than the old version (four calls), but it never falls into infinite recursion for any input. `sum(-5)` returns 0 directly.

3.5.3 Sequences

Many number sequences are defined by a rule that computes each term from the earlier terms. Such a rule is called a **recurrence relation**. A recurrence translates almost word for word into a recursive method: the initial values become the base cases and the relation itself becomes the recursive step. This section treats four sequences. In the Fibonacci and Lucas sequences the method calls itself twice. In the Catalan and Bell sequences the calls are made inside a loop.

3.5.3.1 The Fibonacci sequence

Definition 3.5.4

The **Fibonacci sequence** starts with 0, followed by 1: F(0) = 0, F(1) = 1. Every later term is the sum of the previous two: F(n) = F(n − 1) + F(n − 2).

The sequence is named after the Italian mathematician Leonardo Fibonacci. In his book *Liber Abaci*, written in 1202, he poses a problem about the breeding of rabbits. Every pair of rabbits bears a new pair each month once it is two months old; how many pairs are there at the end of a year? The number of pairs month by month follows this sequence. The first thirteen terms are given in Table 3.5.3.

n	0	1	2	3	4	5	6	7	8	9	10	11	12
F(n)	0	1	1	2	3	5	8	13	21	34	55	89	144

Table 3.5.3: The first terms of the Fibonacci sequence.

The ratio of two consecutive terms approaches the golden ratio (≈ 1.618) as n grows: 144 / 89 ≈ 1.6180. The definition is easy to translate into Java:

```
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

The base case is `n <= 1`, which covers both initial values in one line: if `n` is 0 the method returns 0, and if it is 1 it returns 1. In other words, the answer is `n` itself. In the recursive step the method calls itself **twice**. In factorial the calls form a straight chain; here every call splits into two branches and the calls form a tree. This kind of recursion is called **tree recursion**.

3.5.3.2 The call tree of fibonacci(4)

In tree recursion the order of the calls matters. Java evaluates an expression from left to right. In the expression `fibonacci(n - 1) + fibonacci(n - 2)`, the left term is computed completely first. The call for the right term is made only after all the subcalls of the left term have finished. The tree is therefore traversed by going down to the bottom of the leftmost branch and then moving to the right.

Order	Call	Depth	Event	Returns
1	`fib(4)`	1	`fib(3)` for the left term	3
2	`fib(3)`	2	`fib(2)` for the left term	2
3	`fib(2)`	3	`fib(1)` for the left term	1
4	`fib(1)`	4	base case	1
5	`fib(0)`	4	base case; `fib(2) = 1 + 0`	0
6	`fib(1)`	3	base case; `fib(3) = 1 + 1`	1
7	`fib(2)`	2	recomputed; `fib(1)` for the left term	1

Order	Call	Depth	Event	Returns
8	`fib(1)`	3	base case	1
9	`fib(0)`	3	base case; `fib(2) = 1 + 0`, `fib(4) = 2 + 1`	0

Table 3.5.4: The order of the calls of `fibonacci(4)`. The depth is the number of `fibonacci` frames on the stack at the time of the call.

Table 3.5.4 lists the nine calls in the order they are made. The stack never goes deeper than four `fibonacci` frames, because the longest branch is 4 → 3 → 2 → 1. But the total number of calls is 9, and `fib(2)` is computed twice, `fib(1)` three times and `fib(0)` twice. Figure 3.5.4 shows two moments of the tree: at the top the moment when the descent first reaches the bottom, and at the bottom the final state in which all calls have returned a value.

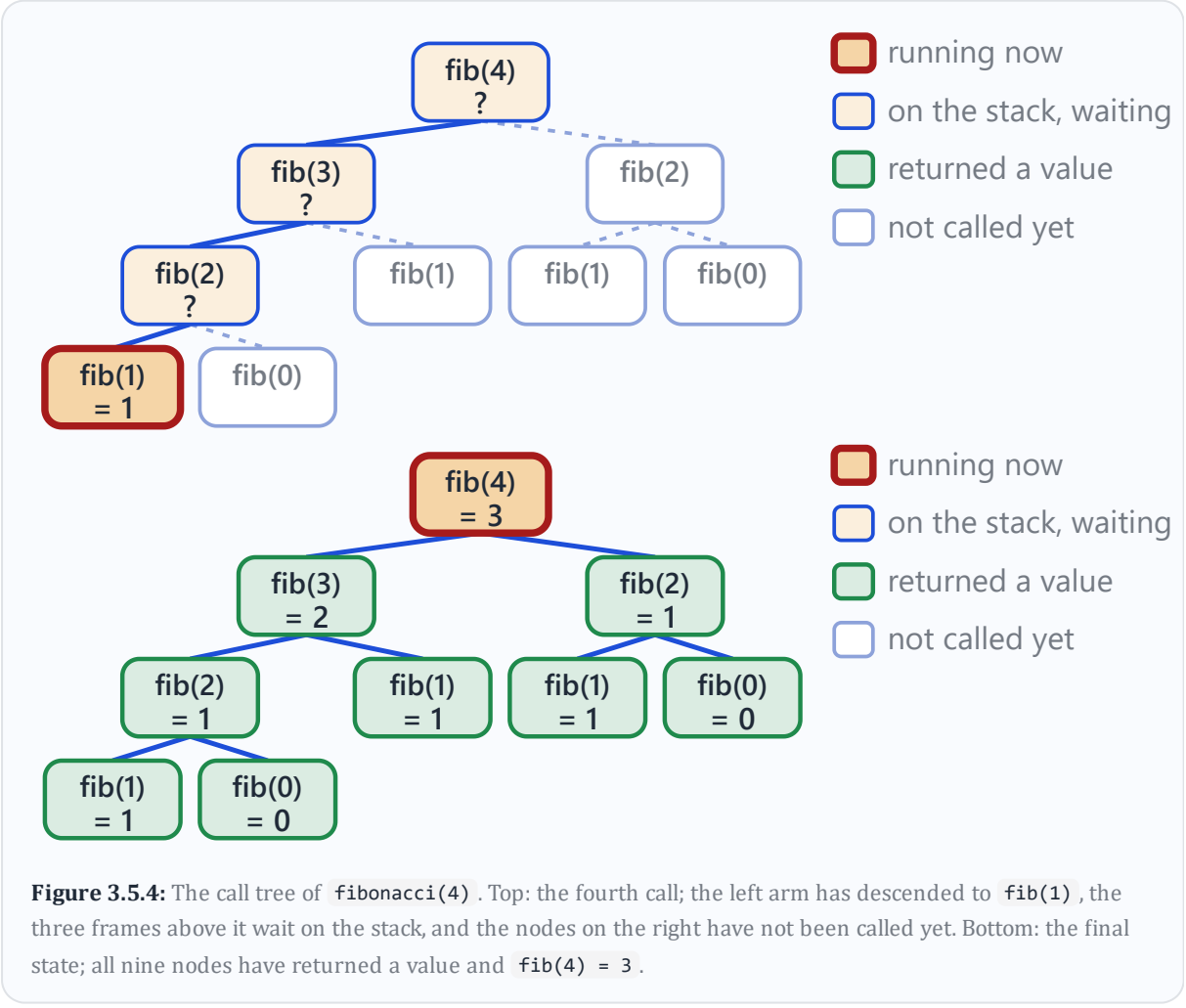


Figure 3.5.4: The call tree of `fibonacci(4)`. Top: the fourth call; the left arm has descended to `fib(1)`, the three frames above it wait on the stack, and the nodes on the right have not been called yet. Bottom: the final state; all nine nodes have returned a value and `fib(4) = 3`.

The repeated computation looks harmless for small n. But the size of the tree grows quickly with n. The number of calls T(n) made by `fibonacci(n)` satisfies the relation T(n) = 1 + T(n − 1) + T(n − 2), and T(n) = 2F(n + 1) − 1. `fibonacci(5)` makes 15 calls, `fibonacci(10)` makes 177 and `fibonacci(40)` makes 331,160,281. The tree widens by a factor of about 1.6 at every level, and the same subtrees are built again and again.

3.5.3.3 The Lucas sequence

The Lucas sequence uses the same relation as Fibonacci with different initial values. It is named after the French mathematician Édouard Lucas.

Definition 3.5.5

The **Lucas sequence** starts with 2, followed by 1: L(0) = 2, L(1) = 1. Later terms are computed as in Fibonacci: L(n) = L(n − 1) + L(n − 2).

```
int lucas(int n) {
    if (n == 0) {
        return 2;
    } else if (n == 1) {
        return 1;
    } else {
        return lucas(n - 1) + lucas(n - 2);
    }
}
```

In Fibonacci the two initial values could be handled in one line with `return n;`, because F(0) = 0 and F(1) = 1. In Lucas, L(0) = 2, so this shortcut does not exist and two separate base cases are needed. The `else if` chain tests `n == 0` first and then `n == 1`; if neither holds, the recursive step runs.

n	0	1	2	3	4	5	6	7	8	9	10
L(n)	2	1	3	4	7	11	18	29	47	76	123

Table 3.5.5: The first terms of the Lucas sequence.

The two sequences are linked: L(n) = F(n − 1) + F(n + 1). For example, L(5) = F(4) + F(6) = 3 + 8 = 11. The call tree of Lucas has the same shape as that of Fibonacci, because the base cases are again at n = 0 and n = 1. So `lucas(5)` also makes 15 calls.

3.5.3.4 The Catalan sequence

Catalan numbers are the answer to many counting problems. The sequence is named after the Belgian mathematician Eugène Catalan.

Definition 3.5.6

The **Catalan sequence** starts with 1: C(0) = 1, C(1) = 1. Its closed formula is C(n) = (2n)! / ((n + 1)! · n!). Its recursive formula is C(n) = Σ C(i) · C(n − 1 − i), where the sum runs from i = 0 to n − 1.

In the recursive formula a term is built from *all* the smaller terms. The Java code computes this sum with a `for` loop and makes the recursive calls inside the loop:

```
long catalan(int n) {
    if (n <= 1) {
        return 1;
    }
    long result = 0;
    for (int i = 0; i < n; i++) {
        result += catalan(i) * catalan(n - i - 1);
    }
    return result;
}
```

The return type is `long`, because Catalan numbers grow quickly. `result` starts at 0, the identity element of addition. The loop advances `i` from 0 to `n - 1` and in each round adds the product `catalan(i) * catalan(n - i - 1)` to the sum. For example, C(3) is computed as follows:

> C(3) = C(0)·C(2) + C(1)·C(1) + C(2)·C(0) = 1·2 + 1·1 + 2·1 = 5

The closed formula gives the same result: 6! / (4! · 3!) = 720 / 144 = 5. The first nine terms of the sequence are 1, 1, 2, 5, 14, 42, 132, 429, 1430; running the code produces these values.

What do Catalan numbers count? The best-known answer is the number of correct arrangements of n pairs of parentheses. In a correct arrangement every opening parenthesis is closed later, and at no point does the number of closing parentheses exceed the number of opening ones. For n = 3 there are five arrangements: `((()))`, `(()())`, `(())()`, `()(())`, `()()()`. The same number also gives the number of ways to parenthesize a product of n + 1 numbers and the number of distinct binary trees with n nodes.

The cost of this method grows even faster than Fibonacci's. Every `catalan(n)` calls all smaller terms twice and stores no result. The measured call counts for n = 2, 3, 4, 5, 6 are 5, 15, 45, 135, 405; whenever n increases by 1, the number of calls triples. A method like this is short and clear for small n, but for large n it needs memoization.

3.5.3.5 The Bell sequence

Definition 3.5.7

The **Bell sequence** starts with 1: B(0) = 1, B(1) = 1. The general formula is B(n) = Σ C(n − 1, k) · B(k), where the sum runs from k = 0 to n − 1. Here C(n, k) is the number of ways to choose k elements out of n, that is, the binomial coefficient.

```
int bell(int n) {
    if (n == 0) { return 1; }
    int count = 0;
    for (int k = 0; k < n; k++) {
        count += choose(n - 1, k) * bell(k);
    }
    return count;
}

int choose(int n, int k) {       // C(n, k)
    return factorial(n) / (factorial(k) * factorial(n - k));
}
```

The code consists of two methods. `bell` calls itself inside a loop as in Catalan, multiplying each term by the binomial coefficient and adding them up. `choose` computes the formula C(n, k) = n! / (k! · (n − k)!) with the `factorial` method. Here one method calls both itself and another recursive method.

Bell numbers count in how many different ways a set of n elements can be divided into groups. For the set {a, b, c} there are five ways: one group `abc`; three ways with one element alone and the other two together, `a|bc`, `b|ac`, `c|ab`; and every element alone, `a|b|c`. Indeed B(3) = 5. The first eight terms of the sequence are 1, 1, 2, 5, 15, 52, 203, 877.

In this code overflow comes from two separate places. `choose` uses the `factorial` that returns `int`, and 13! does not fit in an `int`. `bell(14)` goes wrong because it calls `choose(13, k)`. `bell(13)` = 27,644,437 comes out correctly, but `bell(14)` gives 77,914,768; the correct value is 190,899,322. So the code is correct only for n ≤ 13. Even if `factorial` were fixed, B(16) = 10,480,142,147 would still not fit, because the result is an `int`.

Common mistake

Computing a binomial coefficient as a quotient of factorials. Even when C(n, k) is a small number, the intermediate n! is huge and overflows. A safer way is to use the relation C(n, k) = C(n − 1, k − 1) + C(n − 1, k) (Pascal's triangle) or a loop that multiplies and divides immediately.

3.5.4 Writing a recursive method

Reading a recursive method is easier than writing one. Writing requires obeying two rules and approaching the problem from a particular point of view. This section covers the two rules first, then the error that arises when the rules are broken, and finally the steps of thinking about a problem recursively.

3.5.4.1 Two basic rules

Theorem 3.5.1 (Termination rules)

Rule 1, base case: the method has at least one base case that ends without calling itself. **Rule 2, progress:** every recursive call moves toward a base case; the parameter decreases (or increases) on each call, the string gets shorter, the range gets narrower. If both rules hold, the recursion stops after a finite number of calls for every input.

What progress means is made concrete by a measure called the "distance to the base case". In factorial this measure is `n` itself: it decreases by 1 on every call and the recursion stops when it reaches 1 (Figure 3.5.5). In digit sum the measure is the number of digits, in the palindrome test the length of the string, and in Euclid's algorithm the second parameter. If the measure is a natural number and strictly decreases on every call, it cannot decrease forever; eventually the base case is reached.

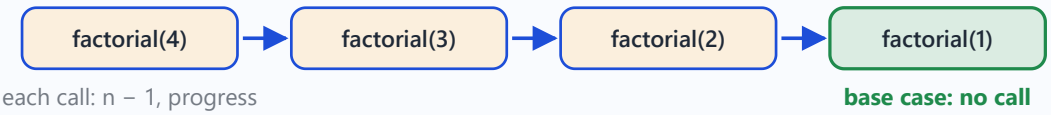


Figure 3.5.5: `factorial(4)` reduces `n` by one on each call: 4, 3, 2, 1. `n = 1` is the base case and no new call is made.

The two most frequent errors are violations of these rules. In **infinite recursion** the base case is either missing or can never be reached. In **insufficient progress** the parameter does not change; for example, `return n * factorial(n);` has been written. The result of both errors is the same: `StackOverflowError`.

Three habits can be added to the rules. First, every recursive call focuses on a smaller subset of the problem. Second, the base case is always tested before the recursive call; if the call is made first, the stack fills up before the test ever gets its turn. Third, all possible inputs are taken into account. Ask whether a negative number, 0, an empty string or `Integer.MIN_VALUE` reaches the base case.

3.5.4.2 Infinite recursion

It is instructive to see what infinite recursion looks like in a complete program. The program below calls `sum`, whose base case is only `n == 1`, with 0:

```
public class Sum {
    static int sum(int n) {
        if (n == 1) {
            return 1;
        } else {
            return n + sum(n - 1);   // line 6
        }
    }
    public static void main(String[] args) {
        System.out.println(sum(0));
    }
}
```

The program compiles; the compiler gives no warning. When it is run, after a while it stops with this message:

```
Exception in thread "main" java.lang.StackOverflowError
        at Sum.sum(Sum.java:6)
        at Sum.sum(Sum.java:6)
        at Sum.sum(Sum.java:6)
        ...
```

Definition 3.5.8

Stack overflow is the call stack filling up in a very deep or infinite recursion. The Java Virtual Machine (JVM) then throws `java.lang.StackOverflowError`, and if it is not caught the program stops.

Let us follow what happens step by step. `main` calls `sum(0)`. For `n = 0`, `0 == 1` is false, and a new call is made to compute `0 + sum(-1)`. `n = -1` is not 1 either; `-1 + sum(-2)` is pending. On every call `n` moves a little further from the base case. Every call adds a new frame to the stack and none of them can return, because each waits for the result of the next. After a thousand calls the stack holds 1001 frames and there is still no error. Finally the memory reserved for the stack runs out (Figure 3.5.6).

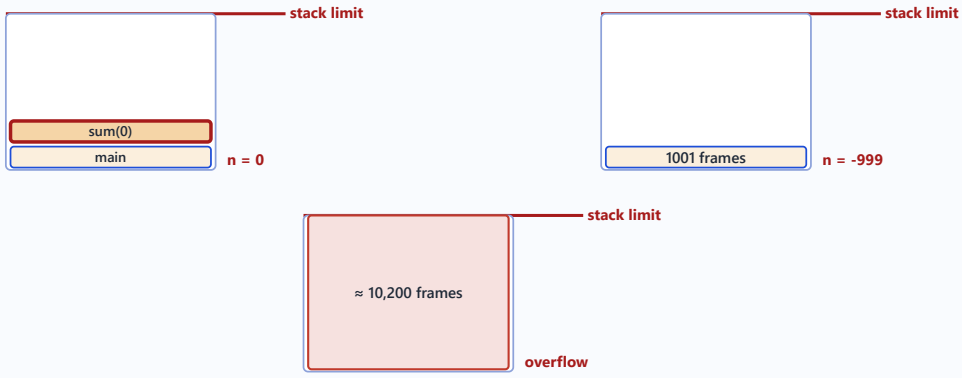


Figure 3.5.6: The stack filling up in the call `sum(0)`. On the left two frames, in the middle 1001 frames, on the right the stack limit has been reached and overflow has occurred. In the measurement shown, the limit was exceeded at ≈ 10,200 calls; the number varies with the JVM settings.

The number of calls after which overflow occurs is not fixed. It depends on the size of the stack, the size of the frame and whether the JVM is currently interpreting the code. In five measurements with JDK 21 on Windows with default settings, overflow occurred between 18,487 and 18,822 calls. When the stack was halved with the `-Xss512k` option, the depth dropped to 6,235. When the same test was repeated within the same JVM, as many as 41,480 calls were possible because the frames of compiled code are smaller. In conclusion, a depth of a few tens of thousands of calls with the default stack is beyond the safe limit.

This error has two important properties. First, the compiler does not catch it; the problem appears only at run time. The compiler sees that every path ends with a `return` and accepts the method as valid; it cannot know whether the values will ever reach the base case. Second, the **stack trace** below the error message gives a familiar picture: the same line (`Sum.java:6`) is repeated hundreds of times one below the other. The JVM prints at most 1024 lines of the stack trace. The same line appearing again and again is the sign of infinite recursion.

The fix is one of three. The base case is widened (`n <= 1` or `n <= 0`). The input is checked at the start of the method and an exception is thrown if it is invalid. If the problem is inherently very deep, for example walking a list of a million items, a loop is used instead of recursion.

Common mistake

Catching `StackOverflowError` with `try-catch` and carrying on. `StackOverflowError` is not an `Exception` but an `Error`; it signals that the program has a bug that must be fixed. Catching it does not prevent the overflow, it only hides the symptom. The right fix is to correct the base case or the progress.

3.5.4.3 Thinking recursively

Four steps are followed when writing a recursive method:

1. **Reduce:** express the problem in terms of a smaller copy of itself.
2. **Base case:** find the smallest input whose answer is known directly.
3. **Trust:** assume the smaller copy is solved correctly; do not unfold the inside of the call in your head.
4. **Combine:** build the larger answer from the smaller one.

The third step is the hardest and is known as the **leap of faith**. When beginners see a recursive call, they try to step into it and unfold the whole chain in their heads. In small examples this works, but in examples such as the Towers of Hanoi it quickly becomes unmanageable. The trust step says this: if, assuming the method works correctly for n − 1, one can show that it also works for n, then the method is correct for every n. This is exactly mathematical induction. The base case is the base step of the induction and the recursive step is the inductive step.

Example 3.5.2

The number of digits of a number is found in four steps. *Reduce:* the number of digits of 48729 is 1 more than the number of digits of 4872. 4872 is obtained from 48729 by dropping the last digit, that is, by the integer division `48729 / 10`. *Base case:* a one-digit number (`n < 10`) has 1 digit. *Trust:* assume that `countDigits(4872)` returns 4. *Combine:* the result is 1 + 4 = 5.

```
int countDigits(int n) {
    if (n < 10) {                       // base case
        return 1;
    }
    return 1 + countDigits(n / 10);   // reduce + combine
}
```

`countDigits(48729)` returns 5, and `countDigits(7)` returns 1 directly. The calls proceed as 48729 → 4872 → 487 → 48 → 4; the last of the five calls is the base case. The code has no `else`. Because the base case ends with `return`, the next line is reached only when `n >= 10`. This style is common in recursive methods: the base case is filtered out first and the remaining body handles the general case.

Note that the method gives wrong results for negative numbers. For −45, `-45 < 10` is true and the result is 1, which is wrong. If negative inputs matter, a sign correction is added at the beginning of the method. How this correction is done can be seen in the digit sum example.

3.5.4.4 Work done on the way down and on the way back

In a recursive method, work can be done in two places: before the method calls itself or after the call returns. The difference is especially clear when the order of the work is visible, as with printing. The two methods below differ only in the order of two lines:

```
static void write(int n) {
    if (n == 0) return;
    System.out.print(n + " ");
    write(n - 1);
}
static void write2(int n) {
    if (n == 0) return;
    write2(n - 1);
    System.out.print(n + " ");
}
static int mystery(int a, int b) {
    if (b == 0) return 0;
    return a + mystery(a, b - 1);
}
```

`write(3)` prints the number and then calls itself. The printing happens on the way down and the output is `3 2 1`. `write2(3)` calls itself first and leaves the printing for the way back. `write2(3)` calls `write2(2)`, which calls `write2(1)`, which calls `write2(0)`. `write2(0)` returns without printing anything. Then the deepest frame, `write2(1)`, prints 1, next `write2(2)` prints 2, and finally `write2(3)` prints 3. The output is `1 2 3`.

Moment	Top of stack	`write` screen	`write2` screen
descent	n = 3	`3`	
descent	n = 2	`3 2`	
descent	n = 1	`3 2 1`	
bottom	n = 0	`3 2 1`	
ascent	n = 1	`3 2 1`	`1`
ascent	n = 2	`3 2 1`	`1 2`
ascent	n = 3	`3 2 1`	`1 2 3`

Table 3.5.6: The moments of printing in the calls `write(3)` and `write2(3)`.

The same code contains a third method, `mystery`. `mystery(4, 3)` expands as follows: 4 + `mystery(4, 2)` = 4 + 4 + `mystery(4, 1)` = 4 + 4 + 4 + `mystery(4, 0)` = 4 + 4 + 4 + 0 = 12. The method computes a × b by adding a to itself b times. The base case is `b == 0` and progress comes from `b` decreasing by 1. But in the call `mystery(4, -1)`, `b` decreases as −1, −2, … and never becomes 0. The result is again a `StackOverflowError`. The second rule requires progress to be *in the direction* of the base case; merely "changing" is not enough.

Work done on the way back is one of the strongest features of recursion. Printing the binary representation of a number is a nice example. The remainders (n % 2) are found starting from the lowest digit, but they must be printed starting from the highest. A method that prints on the way back does this reversal by itself:

```
static void toBinary(int n) {
    if (n > 1) {
        toBinary(n / 2);
    }
    System.out.print(n % 2);
}
```

`toBinary(13)` descends 13 → 6 → 3 → 1. At the bottom, `toBinary(1)` prints 1 % 2 = 1. On the way back, 3 % 2 = 1, 6 % 2 = 0 and 13 % 2 = 1 are printed in turn. The output is `1101` : 13 = 8 + 4 + 1. To do the same job with a loop, one would have to collect the remainders in an array and print them in reverse order. Here the call stack does the work of that array.

3.5.5 Examples

The four examples in this section show different faces of recursion. In the digit sum the problem shrinks by the digits of the number. In the palindrome test the string gets shorter from both ends and there are two separate base cases. In Euclid's algorithm no work is done on the way back. In the Towers of Hanoi the method calls itself twice, and a problem that is hard to write with a loop shrinks to a few lines.

3.5.5.1 Digit sum

The sum of the digits of an integer, for example 4 + 8 + 7 + 2 + 9 = 30 for 48729, is found recursively as follows:

```
int digitSum(int n) {
    if (n < 0) {
        return digitSum(-n);
    } else if (n < 10) {
        return n;
    } else {
        return digitSum(n / 10) + (n % 10);
    }
}
```

The method has three branches. The second and third lines handle a negative number: the sign is flipped and the same method is called again with the positive number. The sum of the digits does not depend on the sign; the digits of −48729 are also 4, 8, 7, 2, 9. The fourth and fifth lines are the base case: the digit sum of a one-digit number is the number itself. The last branch is the general case: the last digit is added to the sum of the remaining digits.

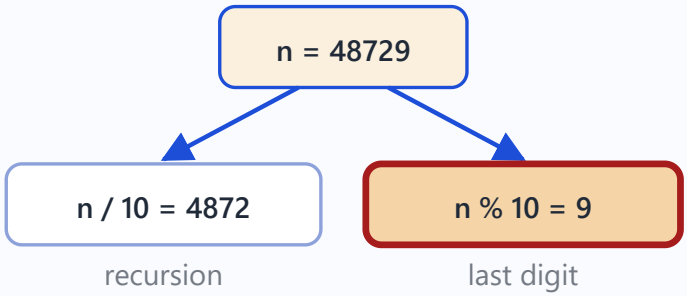


Figure 3.5.7: The number 48729 is split into two parts: `n / 10 = 4872` is processed recursively, and `n % 10 = 9` is the last digit.

The split is done with two integer operations (Figure 3.5.7). Because `n / 10` is a division between integers, it discards the fractional part and gives the number with its last digit removed: 48729 / 10 = 4872. `n % 10` gives the remainder of the division by 10, that is, the last digit: 48729 % 10 = 9. On every call the number gets one digit shorter; progress is guaranteed. The expansion looks like this:

```
digitSum(-48729)
  = digitSum(48729)
  = digitSum(4872) + 9
  = (digitSum(487) + 2) + 9
  = ((digitSum(48) + 7) + 2) + 9
  = (((digitSum(4) + 8) + 7) + 2) + 9
  = (((4 + 8) + 7) + 2) + 9
  = ((12 + 7) + 2) + 9
  = (19 + 2) + 9
  = 21 + 9
  = 30
```

The first call only corrects the sign. In each following step the last digit (9, 2, 7, 8) is left behind as a pending addition. The base case is `digitSum(4)` → 4. On the way back the sums are made from the inside out: 4 + 8 = 12, 12 + 7 = 19, 19 + 2 = 21, 21 + 9 = 30. Six calls are made in total: one sign correction and five digits. In general, a number with d digits takes d calls (d + 1 if it is negative); the time and memory are O(d).

A shorter example, `digitSum(-357)`, is traced together with the call stack in Table 3.5.7. For brevity the method name is written as `r`.

```
static int digitSum(int n) {
    if (n < 0) {
        return digitSum(-n);
    } else if (n < 10) {
        return n;
    } else {
        return digitSum(n / 10) + (n % 10);
    }
}

public static void main(String[] args) {
    System.out.println(digitSum(-357));
}
```

Step	Call at the top	Branch taken	Pending work / returned value
1	`r(-357)`	`n < 0`	calls `r(357)` ; will return its result unchanged
2	`r(357)`	general case	waits for `r(35) + 7`
3	`r(35)`	general case	waits for `r(3) + 5`
4	`r(3)`	`n < 10`	base case, returns 3
5	`r(35)`	return	returns 3 + 5 = 8
6	`r(357)`	return	returns 8 + 7 = 15
7	`r(-357)`	return	returns 15 unchanged
8	`main`	—	prints 15

Table 3.5.7: Trace of the call `digitSum(-357)` .

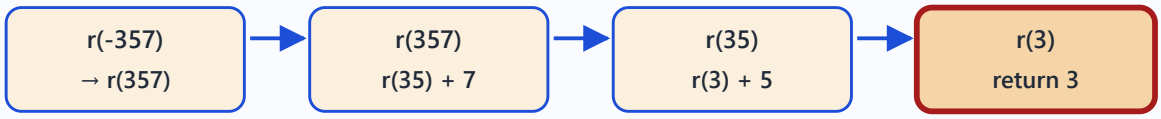


Figure 3.5.8: The deepest moment of the `digitSum(-357)` call chain. `r(3)` is the base case; the frames above it wait for `r(357)` , `r(35) + 7` and `r(3) + 5` respectively.

Four calls are made: one sign correction and three digits. The result is 3 + 5 + 7 = 15 (Figure 3.5.8).

Common mistake

Assuming that negating a number always gives a positive number. The smallest `int` value, `Integer.MIN_VALUE` = −2,147,483,648, has no positive counterpart, because +2,147,483,648 does not fit in an `int` . In Java, `-Integer.MIN_VALUE` again gives −2,147,483,648. For this

reason `digitSum(Integer.MIN_VALUE)` enters the first branch again and again and ends with a `StackOverflowError`. `Math.abs(Integer.MIN_VALUE)` likewise returns a negative number. This single value must be handled separately, or the computation must be done with `long`.

3.5.5.2 Palindrome check

Definition 3.5.9

A **palindrome** is a string that reads the same backward as forward: "kayak", "abba", "ey edip adanada pide ye".

Whether a string is a palindrome can be decided by looking from the outside in. If the first and last characters are equal, then after removing these two characters the remaining string must also be a palindrome (Figure 3.5.9). This is a recursive definition:

```
static boolean isPalindrome(String s) {
    if (s.length() <= 1) {
        return true;
    } else if (s.charAt(0) == s.charAt(s.length() - 1)) {
        return isPalindrome(s.substring(1, s.length() - 1));
    }
    return false;
}
```

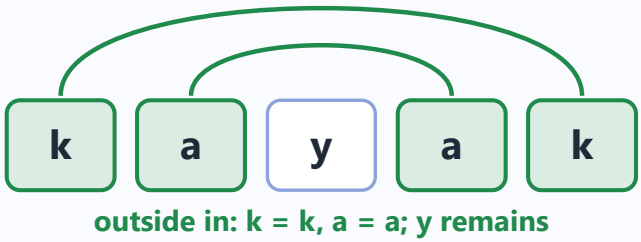


Figure 3.5.9: "kayak" is compared from the outside in: k = k, a = a. The y in the middle is left alone and matches itself.

The code uses two `String` methods. `s.charAt(i)` returns the character at index i of the string; indices start at 0 and the index of the last character is `s.length() - 1`. `s.substring(1, s.length() - 1)` returns the part that starts at index 1 and ends just before index `s.length() - 1`; that is, the string with its first and last letters removed. The second parameter of `substring` is not included in the part.

The method has **two base cases**. If the length is 1 or 0, the string is a palindrome and `true` is returned. The empty string occurs with even-length palindromes: "abba" → "bb" → "" shrinks that way. If the ends differ, the string is not a palindrome and `false` is returned; there is no need to look inside. This second base case is covered by the final line `return false;` when neither the `if` nor the `else if` holds.

Input	Call chain	Result
`"kayak"`	"kayak" → "aya" → "y"	`true`
`"abba"`	"abba" → "bb" → ""	`true`
`"java"`	"java": j ≠ a	`false`
`"Kayak"`	"Kayak": K ≠ k	`false`

Table 3.5.8: A few inputs for `isPalindrome`.

The last row shows an important detail: the comparison is case-sensitive. `'K'` and `'k'` are different characters, and "Kayak" is not counted as a palindrome. "ey edip adanada pide ye", on the other hand, gives `true` because it is symmetric including its spaces. For sentences containing spaces and punctuation, these characters must first be removed and the letters lowercased.

The calls made for the word "racecar" are traced in Table 3.5.9.

Call	`s`	Length	Ends	Next call / return
1	`"racecar"`	7	'r' = 'r'	`s.substring(1, 6)` = `"aceca"`
2	`"aceca"`	5	'a' = 'a'	`s.substring(1, 4)` = `"cec"`
3	`"cec"`	3	'c' = 'c'	`s.substring(1, 2)` = `"e"`
4	`"e"`	1	—	base case: returns `true`

Table 3.5.9: Trace of the call `isPalindrome("racecar")`.

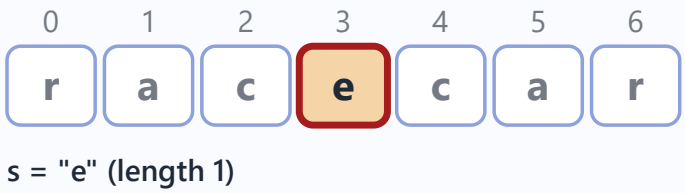


Figure 3.5.10: The part examined in the fourth call is only the "e" in the middle. Since length 1 ≤ 1, the base case runs.

The fourth call is the base case (Figure 3.5.10). On the way back each frame passes the `true` it receives upward unchanged; there is no other work to do on the way back. Result: "racecar" is a palindrome.

In terms of cost this method is more expensive than it first appears. For a string of length n about n/2 calls are made. But the `substring` in each call creates a new string object in Java and copies the characters. The total number of characters copied is $(n - 2) + (n - 4) + \ldots \approx n^2/4$. So the time is $O(n^2)$, and the memory is also $O(n^2)$ because of the pending strings. A version that carries two indices instead of the string, for example `isPalindrome(s, lo, hi)`, avoids this copying and runs in O(n) time.

The indexed version looks like this:

```
static boolean isPalindrome(String s, int lo, int hi) {
    if (lo >= hi) {
        return true;
    } else if (s.charAt(lo) != s.charAt(hi)) {
        return false;
    }
    return isPalindrome(s, lo + 1, hi - 1);
}
```

The string never gets shorter; instead the bounds of the part being examined narrow. `lo` is the index of the first character of the part and `hi` the index of its last. The first call is made as `isPalindrome(s, 0, s.length() - 1)`. The base cases follow the same logic. If `lo >= hi`, at most one character is left in the part and the result is `true`. If the ends differ, the result is `false`. Progress comes from the two indices approaching each other: the gap shrinks by 2 on every call. For "racecar" the calls are (0, 6), (1, 5), (2, 4), (3, 3), and the fourth call is the base case. For the empty string the first call is (0, −1) and it returns `true` immediately. This technique is common in recursive methods that work on strings and arrays: instead of copying the data, carry the bounds as parameters.

3.5.5.3 Greatest common divisor

The **greatest common divisor** (GCD) of two positive integers is the largest number that divides both without a remainder. In the code the method is called `gcd` after its initials. Euclid's algorithm, about 2300 years old, finds the GCD with division remainders:

> gcd(x, y) = gcd(y, x mod y), gcd(x, 0) = x

```
static int gcd(int x, int y) {
    if (y == 0) {
        return x;
    }
    return gcd(y, x % y);
}
```

The base case is `y == 0`: the GCD of a number and 0 is the number itself. In the general case `y` takes the place of `x` and `x % y` takes the place of `y`. The correctness of the relation rests on this: every number that divides both x and y also divides x − k·y, and therefore also divides x mod y. The set of common divisors stays the same at every step; only the numbers get smaller.

Progress comes from the remainder getting smaller. x mod y is always smaller than y; the second parameter strictly decreases on every call and finally reaches 0. The order of the parameters does not matter either. In the call `gcd(93, 444)`, since 93 % 444 = 93, the first step only swaps the two numbers and gives `gcd(444, 93)`.

```
gcd(444, 93)
  = gcd(93, 72)     // 444 % 93 = 72
  = gcd(72, 21)     // 93 % 72 = 21
  = gcd(21, 9)      // 72 % 21 = 9
  = gcd(9, 3)       // 21 % 9 = 3
  = gcd(3, 0)       // 9 % 3 = 0
  = 3
```

Call	x	y	x % y	Event
`gcd(444, 93)`	444	93	72	444 = 4 · 93 + 72
`gcd(93, 72)`	93	72	21	93 = 1 · 72 + 21
`gcd(72, 21)`	72	21	9	72 = 3 · 21 + 9
`gcd(21, 9)`	21	9	3	21 = 2 · 9 + 3
`gcd(9, 3)`	9	3	0	9 = 3 · 3 + 0
`gcd(3, 0)`	3	0	—	base case: returns 3

Table 3.5.10: Trace of the calls of `gcd(444, 93)`.

Six calls are made and gcd(444, 93) = 3 is found (Table 3.5.10). This method differs from the earlier ones on the way back. In the line `return gcd(y, x % y);` there is no work left to do after the call returns. The 3 found by the last call passes unchanged through every frame and reaches `main`. Such calls are called tail calls. Another example: `gcd(48, 18)` → `gcd(18, 12)` → `gcd(12, 6)` → `gcd(6, 0)` → 6.

Euclid's algorithm is very fast. In every two steps the first number at least halves; so the number of calls is proportional to the number of digits of the numbers, that is, O(log min(x, y)). The worst case occurs with two consecutive Fibonacci numbers; for example `gcd(89, 55)` makes ten calls.

3.5.5.4 Towers of Hanoi

The Towers of Hanoi is a puzzle introduced by Édouard Lucas in 1883. There are three pegs (A, B, C), and peg A holds n disks stacked from largest to smallest. All the disks are to be moved to C. There are two rules: on each move only one disk, the topmost disk of a peg, is moved; and a larger disk is never placed on top of a smaller one.

The puzzle is hard when thought of with loops, but when thought of recursively the solution appears by itself. The largest disk can be moved to C only when the n − 1 disks above it have been moved somewhere else, that is, to the auxiliary peg B. So the solution has three steps: move the top n − 1 disks to the auxiliary peg, move the largest disk to the target, then move the n − 1 disks from the auxiliary peg to the target. The first and third steps are the same problem with one disk fewer. The base case is a single disk: it is moved directly.

```
public class TowersOfHanoi {
    static void solve(int n, char from,
                      char to, char via) {
        if (n == 1) {   // one disk: move it
            System.out.println(n + ": " + from + "->" + to);
        } else {
            solve(n - 1, from, via, to);
            System.out.println(n + ": " + from + "->" + to);
            solve(n - 1, via, to, from);
        }
    }
    public static void main(String[] args) {
        solve(3, 'A', 'C', 'B');
    }
}
```

```
1: A->C
2: A->B
1: C->B
3: A->C
1: B->A
2: B->C
1: A->C
```

The `solve` method takes four parameters: the number of disks and the names of the source, target and auxiliary pegs. The peg names are of type `char`. In the recursive calls the roles of the pegs change. In the first call, `solve(n - 1, from, via, to)` is written: n − 1 disks are moved from the source to the *auxiliary* peg, and during this the real target plays the role of the auxiliary. In the call on the third line, `solve(n - 1, via, to, from)` is written: n − 1 disks are moved from the auxiliary peg to the target, and this time the source, being empty, serves as the auxiliary. The number in the printed line is the number of the disk being moved; the smallest disk is 1.

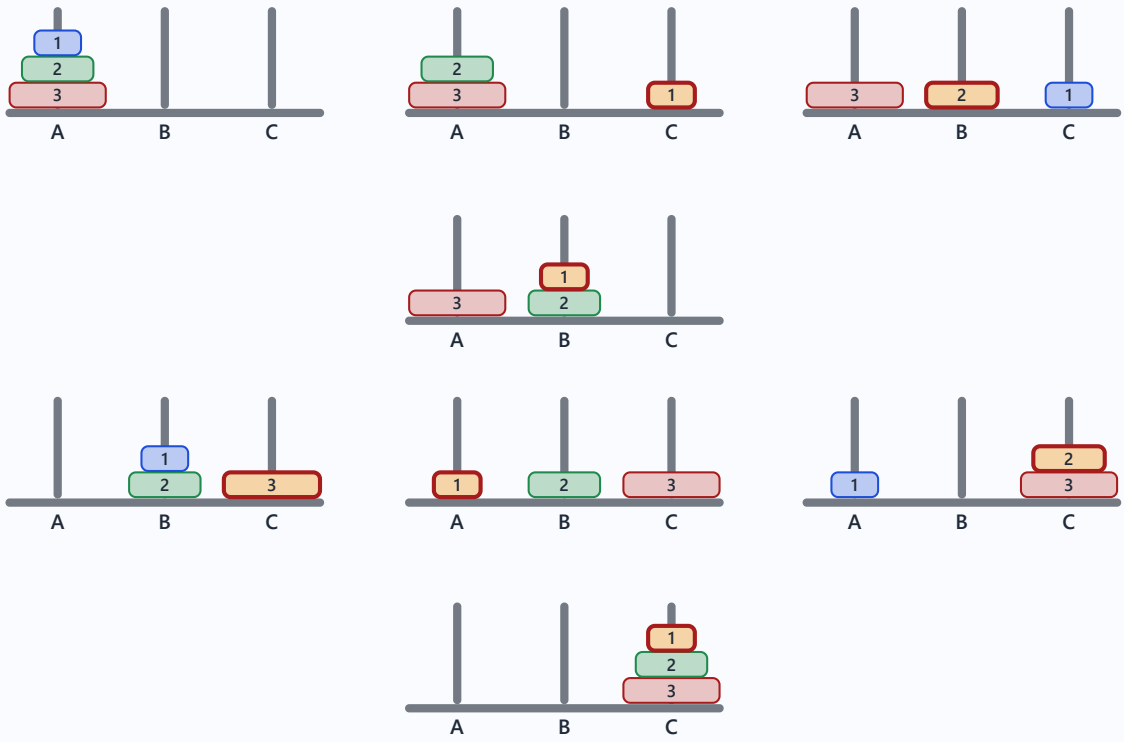


Figure 3.5.11: `solve(3, 'A', 'C', 'B')` : the initial state and the state after each of the seven moves (left to right, top to bottom). After the third move the top two disks are on B; the fourth move carries the largest disk to C.

Move	Output	Printing call	Meaning
1	`1: A->C`	`solve(1, A, C, B)`	base case
2	`2: A->B`	`solve(2, A, B, C)`	disk 2 to the auxiliary peg
3	`1: C->B`	`solve(1, C, B, A)`	the top 2 disks are now on B
4	`3: A->C`	`solve(3, A, C, B)`	largest disk to the target
5	`1: B->A`	`solve(1, B, A, C)`	the 2 disks start moving to C
6	`2: B->C`	`solve(2, B, C, A)`	disk 2 onto disk 3
7	`1: A->C`	`solve(1, A, C, B)`	puzzle solved

Table 3.5.11: Which call each move of `solve(3, 'A', 'C', 'B')` comes from.

The moves are given in Table 3.5.11 together with the call that makes each. The first three moves are the work of `solve(2, A, B, C)`; the fourth move is the middle line of `solve(3, …)`; the last three moves are the work of `solve(2, B, C, A)` (Figure 3.5.11). The call stack never goes deeper than three `solve` frames; the stack depth equals the number of disks.

Theorem 3.5.2 (Number of Hanoi moves)

The Towers of Hanoi with n disks can be solved in at least $2^n - 1$ moves, and the `solve` method makes exactly that many moves.

The number of moves H(n) can be read off the structure of the method: two solutions for n − 1 disks and one move. $H(n) = 2 \cdot H(n-1) + 1$ and $H(1) = 1$. From this $H(2) = 3$, $H(3) = 7$, $H(4) = 15$, and in general $H(n) = 2^n - 1$. This number grows very fast. For 10 disks 1023 moves are needed. For the 64 disks of the legend, $2^{64} - 1 \approx 1.8 \times 10^{19}$ moves are needed; at one move per second this takes about 585 billion years. The running time of the method is also $O(2^n)$, but this time nothing is wasted: every move is necessary and no shorter solution exists.

An interesting property of the puzzle is that the first move depends on the number of disks. The measured first moves are `1: A->C` when n is odd and `1: A->B` when n is even. For example, for 4 disks 15 moves are made and the first move carries the smallest disk to the auxiliary peg. Solutions of Hanoi written with loops rely on observations of this kind and are much longer than the recursive version. The recursive version is only a few lines because it mirrors the structure of the problem as it is.

3.5.6 Efficiency

A recursive solution can be short and clear, but that does not mean it is fast or light. This section gathers the price of recursion under three headings: the cost of each call, the stack depth and the repeated computation of the same values. It then shows how recursive methods are converted into loops.

3.5.6.1 The cost of one call

On every method call the JVM does some work: a new frame is opened, the parameters are copied, and the return address is recorded. On return the frame is removed and the value is carried to the frame below. In a loop iteration, on the other hand, only a few variables are updated. For this reason a recursive method doing the same job is usually a little slower than a loop.

The more important difference is in memory. As the recursive calls are made, the pending partial results, that is, the half-finished work such as `4 * ?` in each frame, accumulate on the call stack. Table 3.5.12 summarizes the three main costs.

Cost	Description	Example
Memory	Every pending call holds a frame.	`factorial(n)` : n frames, O(n) memory; O(1) with a loop.
Depth limit	When the stack is full, a `StackOverflowError` occurs.	After a depth of a few tens of thousands of calls with the default stack.

Cost	Description	Example
Repetition	Tree recursion may compute the same value many times.	`fibonacci(40)` : 331,160,281 calls.

Table 3.5.12: The three costs of recursion.

Definition 3.5.10

An **iterative solution** is one that does the repetition with a loop instead of recursion. It does not pile up partial results on the stack; it keeps them in an accumulator variable and runs in a single frame.

In problematic cases the remedy is to switch to the iterative version. Factorial is the simplest example: `factorial(100000)` in the recursive version, overflow of the result aside, most likely ends with a `StackOverflowError`. In a trial with JDK 21 this is indeed what happened. The loop version finishes the same call in a single frame and in under a second; although its result is wrong, it never suffers a stack overflow.

3.5.6.2 Repeated computation

The real problem of tree recursion is not memory but time. Figure 3.5.12 shows the call tree of `fibonacci(5)`. The tree has 15 nodes. The red nodes are recomputations of values that were already computed: `fib(3)` is computed twice and `fib(2)` three times.

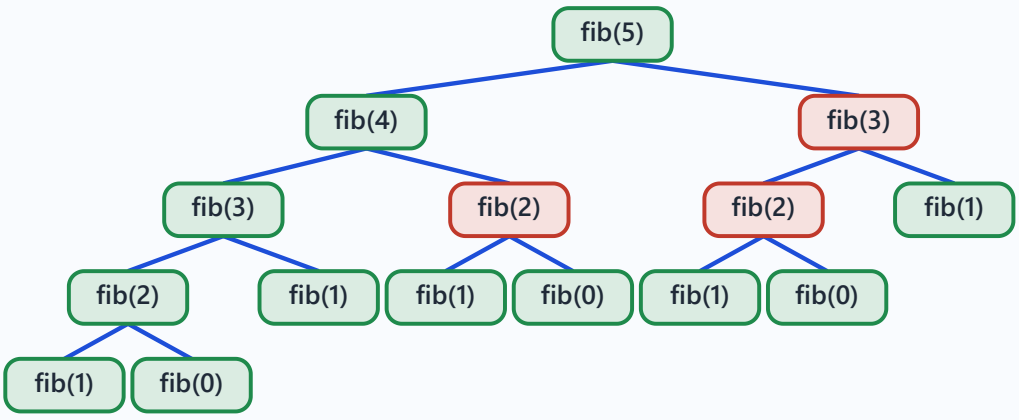


Figure 3.5.12: The call tree of `fibonacci(5)` : 15 calls. The red nodes are recomputations of values that were already computed.

The repetition grows exponentially as n increases. The number of calls is given by the formula T(n) = 2F(n + 1) − 1 and increases by a factor of about 1.6 whenever n goes up by 1; this ratio is the golden ratio. Table 3.5.13 gives the call counts and times measured with JDK 21.

n	Number of calls	F(n)	Measured time
10	177	55	< 0.1 ms
20	21,891	6,765	≈ 0.3 ms
30	2,692,537	832,040	≈ 5 ms
40	331.160.281	102.334.155	≈ 0,5 s
45	3.672.623.805	1.134.903.170	≈ 11 s

Table 3.5.13: Number of calls and measured time for `fibonacci(n)` without memoization.

In the table, every ten steps multiply the number of calls by about 120. `fibonacci(40)` takes half a second. At this rate `fibonacci(50)` reaches two minutes; that is, it takes minutes. One more Java-specific problem is added to this code: F(47) = 2,971,215,073 exceeds the limit of `int`. After running for two minutes, `fibonacci(50)` returning `int` returns a meaningless number such as −298632863. The correct value is 12,586,269,025.

The source of the problem is that the method forgets the values it has already computed. The number of distinct values `fibonacci(40)` needs is only 41 (from F(0) to F(40)). If each were computed once and stored, a few dozen operations would be enough. There are two ways to do this: converting to a loop and memoization.

3.5.6.3 Converting to a loop

Linearly recursive methods, that is, methods that call themselves once, can usually be converted into loops easily. The method is this: the work done on the way back is done in the loop with an accumulator variable. The two versions of factorial side by side look like this:

```
int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

```
int factorial(int n) {
    int product = 1;
    for (int i = 1; i <= n; i++) {
        product *= i;
    }
    return product;
}
```

In the loop version `product` starts at 1, the identity element of multiplication. The loop advances `i` from 1 to `n` and in each round multiplies `product` by `i`. For `factorial(5)` `product` takes the values 1, 1, 2, 6, 24, 120. For `n = 0` the loop never runs and 1 is returned; 0! = 1 is covered without needing to be written separately. The two versions give the same result, but their memory use differs (Figure 3.5.13). Both versions assume `n` ≥ 0. For a negative `n` the recursive version never reaches the base case and throws `StackOverflowError`; the loop version never enters the loop and silently returns 1. Also, 13! = 6,227,020,800 does not fit in an `int`; the `int` versions are correct only up to 12!.



Figure 3.5.13: The recursive `factorial(5)` opens five frames; the loop version updates the variable `product` in a single frame.

Digit sum and palindrome are converted into loops in the same way:

```
int digitSum(int n) {
    int sum = 0;
    int rest = Math.abs(n);
    while (rest > 0) {
        sum += rest % 10;
        rest /= 10;
    }
    return sum;
}
```

`rest` starts with the absolute value of the number. In each round the last digit is added to `sum` and removed from `rest`. For 48729 `sum` goes 9, 11, 18, 26, 30; `rest` goes 4872, 487, 48, 4, 0 and the loop ends. `digitSum(-48729)` returns 30, the same as the recursive version. The sign is removed with `Math.abs`, because the sum of digits cannot be negative. This version does not suffer a stack overflow for `Integer.MIN_VALUE`, but because `Math.abs` returns a negative value it never enters the loop and gives 0; the correct answer is 47.

```
boolean isPalindrome(String s) {
    s = s.toLowerCase();
    int len = s.length();
    for (int i = 0; i < len / 2; i++) {
        if (s.charAt(i) != s.charAt(len - 1 - i)) {
            return false;
        }
    }
    return true;
}
```

The loop palindrome uses indices that move inward from both ends instead of shortening the string. `i` advances from the start and `len - 1 - i` from the end. The loop runs up to half the string; for an odd length the middle character matches itself and is not examined. It returns `false` at the first differing pair. If no differing pair is found, the loop ends and `true` is returned. Because this version

calls `toLowerCase()` at the start, it counts "Kayak" as a palindrome too. In real programs `toLowerCase(Locale.ROOT)` should be preferred: if the default locale is `tr`, the letter `I` is converted not to `i` but to `ı`.

The trace for "Kepek" is given in Table 3.5.14.

```
static boolean isPalindrome(String s) {
    s = s.toLowerCase();
    int len = s.length();
    for (int i = 0; i < len / 2; i++) {
        if (s.charAt(i) != s.charAt(len - 1 - i)) {
            return false;
        }
    }
    return true;
}

public static void main(String[] args) {
    System.out.println(isPalindrome("Kepek"));
}
```

Step	`s`	`len`	`i`	Comparison	Result
1	`"Kepek"`	—	—	—	parameter
2	`"kepek"`	—	—	—	`toLowerCase()` returns a new string
3	`"kepek"`	5	—	—	loop bound 5 / 2 = 2
4	`"kepek"`	5	0	`s.charAt(0)` 'k', `s.charAt(4)` 'k'	equal
5	`"kepek"`	5	1	`s.charAt(1)` 'e', `s.charAt(3)` 'e'	equal
6	`"kepek"`	5	2	`2 < 2` is false	loop ends
7	`"kepek"`	5	—	—	returns `true` and is printed

Table 3.5.14: Trace of the loop-based `isPalindrome("Kepek")`.

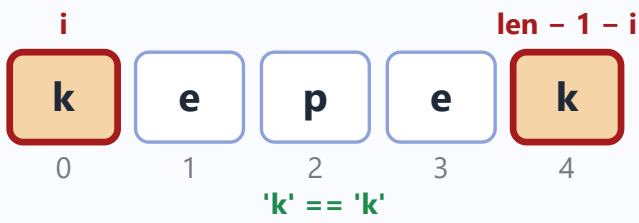


Figure 3.5.14: The indices compared in the round `i = 0`: index 0 from the start, index 5 − 1 − 0 = 4 from the end. Both characters are 'k'.

The 'p' in the middle is never compared; that it matches itself is certain anyway (Figure 3.5.14). A single frame is used, and, unlike the recursive version, a new string is not produced at every step. The time is O(n), and the extra memory is O(1) apart from the single copy produced by `toLowerCase()`.

Tree recursion can also be converted into a loop, but this time the accumulator is not a single variable. In Fibonacci each term depends on the previous *two* terms; the loop carries those two terms:

```
static long fibonacci(int n) {
    long a = 0, b = 1;
    for (int i = 0; i < n; i++) {
        long next = a + b;
        a = b;
        b = next;
    }
    return a;
}
```

`a` and `b` are two consecutive Fibonacci numbers, starting with F(0) = 0 and F(1) = 1. In each round both advance one step: the new `a` becomes the old `b`, and the new `b` becomes the old sum. If written without the temporary variable `next`, the old `a` would be lost. After n rounds `a` = F(n). For `fibonacci(10)` the pair `(a, b)` goes (0, 1), (1, 1), (1, 2), (2, 3), (3, 5), (5, 8), (8, 13), (13, 21), (21, 34), (34, 55), (55, 89) and 55 is returned. This version runs n rounds: time O(n), memory O(1). It gives `fibonacci(50)` = 12586269025 and `fibonacci(90)` instantly. The work the recursive version does with 331 million calls, this loop does in 40 rounds.

Common mistake

Writing the loop bound as `i <= len / 2` or `i < len`. The first form does not change the result but adds a needless round. With the second, every pair is compared twice. The real danger is writing the opposite index as `len - i`: at `i = 0`, `s.charAt(5)` is called and a `StringIndexOutOfBoundsException` is thrown. The index of the last character is `len - 1`.

3.5.6.4 Recursion or a loop?

The two approaches are two ways of writing the same repetition. Table 3.5.15 summarizes the differences between them.

	Recursion	Loop (iterative)
Stopping	Base case	Loop condition
Progress	Call with a smaller argument	Updating a counter or variable
Memory	One frame per call: O(depth)	Usually O(1)
Speed	Cost of calls; repetition in tree recursion	Usually faster
Failure	`StackOverflowError` if there is no base case	Infinite loop if the condition is broken (the program hangs)

	Recursion	Loop (iterative)
Readability	Very natural for trees, divide and conquer, Hanoi, sequences	Natural for simple counting and accumulating jobs

Table 3.5.15: Recursion compared with a loop.

The difference in the failure row is interesting. A broken loop never ends and the program hangs; it has to be stopped from outside. A broken recursion, on the other hand, fills the stack quickly and stops with an error message. For finding the bug the second is easier, because the stack trace shows where the problem is.

Every recursive solution can be written with a loop, and the converse is also true. For linear recursion the conversion is easy with an accumulator. For tree recursions such as Hanoi, however, the loop version has to imitate the work of the call stack with a stack data structure that it maintains itself, and the code gets longer. The choice is made by three criteria. If clarity of the code favors recursion and the depth is small (the number of disks in Hanoi, the height of a balanced tree), recursion is appropriate. If the depth grows linearly with the input (a list of a million items), a loop is chosen. If performance is critical and the tree recursion contains repetition, a loop or memoization is needed.

3.5.7 Tail recursion

In recursive methods the memory problem comes from work that waits on the stack to be done on the way back. If no work remains for the way back, the frames need not wait either. Tail recursion is based on this idea.

3.5.7.1 The tail call and the accumulator

Definition 3.5.11

Tail recursion is recursion in which the last operation of the method is the recursive call itself; no work remains after the call returns. The result is carried in an extra **accumulator** parameter.

The distinction can be seen in one line. The recursive lines of ordinary factorial and of tail-recursive factorial are as follows:

```
return n * factorial(n - 1);
// multiply AFTER the call returns
```

```
return factorial(n - 1, n * acc);
// multiply first; the call is last
```

In the left line `factorial(n - 1)` is called first, and *after* it returns the result is multiplied by `n`. The multiplication is pending work, and so the value of `n` has to be kept in the frame. In the right line the multiplication is done *before* the call, while the argument is being computed. The value `n * acc` is sent to the new call as a parameter. When the call returns, the value that comes back is returned as it is; there is no other work to do (Figure 3.5.15).

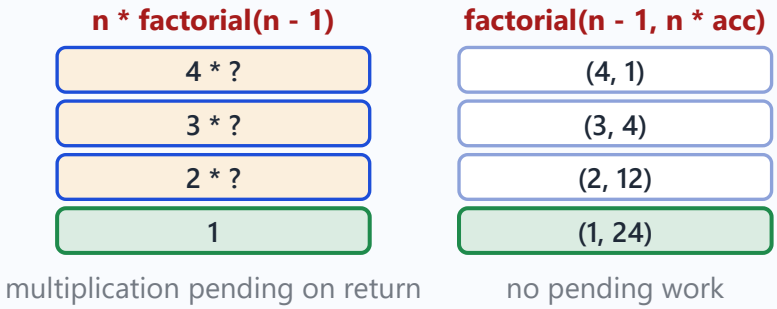


Figure 3.5.15: Left: ordinary recursion; every frame holds a product to be computed on the way back. Right: tail recursion; the frames only carry arguments and there is no pending work.

This form makes **tail call optimization** (TCO) possible for the compiler. Since there is no pending work, the new call can reuse the old frame; the call is handled like a loop and frames do not pile up. Memory drops from O(n) to O(1) and the depth limit disappears.

Java, however, does not perform this optimization. Neither the `javac` compiler nor the JVM turns tail calls into loops. Tail-recursive Java code still opens a frame for every call and uses O(n) stack; the depth limit applies as before. Tail call optimization is mandatory in Scheme. In Scala the compiler turns self-calling tail recursions into loops, and the `@tailrec` annotation checks this. In Kotlin, methods marked with the `tailrec` keyword are turned into loops. In Java the value of tail recursion is that it makes conversion to a loop easier: a tail recursion with an accumulator can be turned into a `while` loop almost mechanically.

Common mistake

Taking a call that sits at the end of a line to be a tail call. In the line `return 1 + countDigits(n / 10);` the call is at the far right, but 1 is added on the way back; it is not a tail call. In the line `return reverse(s.substring(1)) + s.charAt(0);` concatenation is also done on the way back. The criterion is not where the call is written but whether any work remains after the call.

3.5.7.2 Examples and the wrapper method

A linearly recursive method is made tail-recursive by adding an accumulator parameter. For digit sum and factorial the result looks like this:

```
int digitSum(int num, int sum) {
    if (num == 0) {
        return sum;
    } else {
        return digitSum(num / 10,
                        sum + num % 10);
    }
}
```

The last digit is added to `sum` before the call. `digitSum(48729, 0)` proceeds as follows: (48729, 0) → (4872, 9) → (487, 11) → (48, 18) → (4, 26) → (0, 30). When `num == 0`, the 30 in the accumulator is returned. This version is correct only for `n ≥ 0`; `digitSum(-357, 0)` returns −15, because Java gives a negative remainder for negative numbers.

```
int factorial(int n, int acc) {
    if (n == 0) {
        return acc;
    } else {
        return factorial(n - 1, n * acc);
    }
}
```

In factorial the product is folded into `acc` before the call, and `factorial(5, 1)` returns 120. The base case is now `n == 0`; at that point the result is ready in the accumulator.

The initial value of the accumulator is the identity element of the operation: 0 for addition, 1 for multiplication. This detail should not be a burden on whoever uses the method. For this reason a one-parameter **wrapper method** that hides the accumulator is written:

```
static int factorial(int n) {
    return factorial(n, 1);
}
```

The two methods have the same name but different numbers of parameters. In this case Java looks at the number of arguments in the call to tell which one is meant; this feature is called method overloading. The user writes `factorial(5)` and never sees the accumulator.

Here is a counterexample: the `reverse` method reverses a string.

```
static String reverse(String s) {
    if (s.isEmpty()) return "";
    return reverse(s.substring(1)) + s.charAt(0);
}
```

`reverse("java")` is computed as `reverse("ava") + 'j'`; that in turn needs `reverse("va") + 'a'`, and so on. The result is `"avaj"`. This method is not tail-recursive, because concatenation (`+`) is done after the call returns. `gcd`, on the other hand, is tail-recursive: its last action is the call itself.

The tail-recursive version of Fibonacci carries the two variables of the loop version as parameters:

```
static long fibAcc(int n, long a, long b) {
    if (n == 0) {
        return a;
    }
    return fibAcc(n - 1, b, a + b);
}
```

`a` and `b` are again two consecutive terms, and the first call is `fibAcc(n, 0, 1)`. Each call decreases `n` by 1 and shifts the pair by one step. When `n` has dropped to 0, `a` is the requested term. `fibAcc(10, 0, 1)` returns 55 and makes only 11 calls. Compared with the tree-recursive version the difference is striking: the same relation is computed with 177 calls or with 11, depending on how the result is carried. The loop version and the tail-recursive version match line by line. The loop variables correspond to the parameters, the loop condition to the opposite of the base case, and one round to one call.

3.5.7.3 Tracing factorial(4, 1)

```
static int factorial(int n, int acc) {
    if (n == 0) {
        return acc;
    } else {
        return factorial(n - 1, n * acc);
    }
}
```

Call	`n`	`acc`	Work before the call	Work on the way back
1	4	1	4 * 1 = 4 → `factorial(3, 4)`	returns 24 unchanged
2	3	4	3 * 4 = 12 → `factorial(2, 12)`	returns 24 unchanged
3	2	12	2 * 12 = 24 → `factorial(1, 24)`	returns 24 unchanged
4	1	24	1 * 24 = 24 → `factorial(0, 24)`	returns 24 unchanged
5	0	24	base case	returns 24

Table 3.5.16: Trace of the call `factorial(4, 1)`.

All the multiplications are done on the way down (Table 3.5.16). When the base case is reached, the result is already in `acc`. No computation is done on the way back; 24 is simply passed upward. Compared with ordinary factorial, the order of the multiplications is also reversed: here they are

multiplied in the order 4, 3, 2, 1.

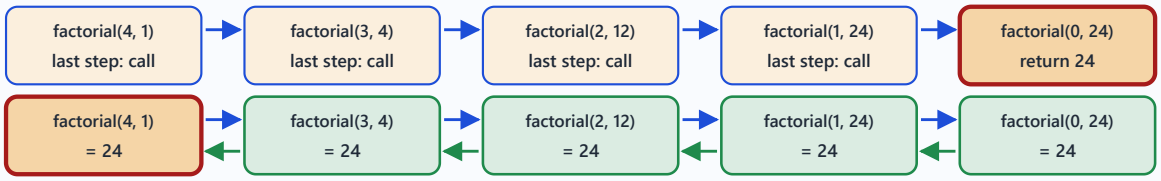


Figure 3.5.16: The call chain of `factorial(4, 1)`. Top: the base case; the result is ready in `acc`. Bottom: all frames have returned 24 unchanged.

Figure 3.5.16 shows an important fact: because Java does not optimize tail calls, all five `factorial` frames still go onto the stack. In a language that performs tail call optimization, a single frame would be enough. The loop equivalent of the same method in Java is: `while (n != 0) { acc = n * acc; n = n - 1; } return acc;`. The parameters become variables, and the recursive call turns into updates of these variables.

3.5.8 Memoization

One way to remove the repetition in tree recursion was to switch to a loop. The second way is to keep the recursive structure and give the method a memory. When the method computes a value for the first time, it writes it down somewhere; if the same value is requested again, it reads it instead of computing.

Definition 3.5.12

Memoization is a method storing the results it has computed and returning them without recomputation when the same argument comes again. The structure that holds the results is called a **cache**. In this way the method carries state between calls.

3.5.8.1 Fibonacci with memoization

The easiest tool for a cache in Java is a `Map`. A `Map` is a data structure that maps keys to values; here the key is `n` and the value is F(n). The notation `Map<Integer, Long>` says that the keys are `Integer` and the values are `Long`. Because a `Map` cannot hold primitive types, their wrapper classes are used instead of `int` and `long`; Java does the conversion automatically.

```
import java.util.HashMap;
import java.util.Map;

static Map<Integer, Long> memo = new HashMap<>();

static long fibonacci(int n) {
    if (memo.containsKey(n)) {
        return memo.get(n);    // computed before
    }
    if (n == 0 || n == 1) {
        return n;
    }
    long result = fibonacci(n - 1) + fibonacci(n - 2);
    memo.put(n, result);        // store the result
    return result;
}
```

The body of the method has four steps. First the memory is consulted: `memo.containsKey(n)` returns `true` if the key `n` was written before, and the value is read with `memo.get(n)` and returned. If it is not in memory, the base case is tested. If neither applies, the value is computed as usual with two recursive calls. Finally `memo.put(n, result)` stores the result, and the result is returned.

`memo` is defined as a field, outside the method, and `static`. Were it a local variable, it would be created anew and empty on every call, and nothing would be remembered. For all calls to share the same memory it must be kept outside the method. The return type has also been changed to `long`. Because memoization makes large n quickly computable, `int` is no longer enough: `fibonacci(90)` instantly returns 2880067194370816120. The largest Fibonacci number that fits in a `long` is F(92) = 7540113804746346429.

Because the keys are consecutive integers starting from 0, an array can be used instead of a `Map`:

```
static long[] table = new long[93];

static long fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    if (table[n] != 0) {
        return table[n];
    }
    table[n] = fibonacci(n - 1) + fibonacci(n - 2);
    return table[n];
}
```

The length of the array is 93, because the last term that fits in a `long` is F(92). Java fills all elements of a new `long` array with 0. In this version 0 means "not computed yet". This convention is safe, because for n ≥ 2 no Fibonacci number is 0. The array uses less memory than the `Map` and is faster. No wrapper objects are created, and the key lookup reduces to a single index access. The price is flexibility: the length of the array must be known in advance and the keys must be small, non-negative integers. `fibonacci(90)` also returns 2880067194370816120 in this version.

3.5.8.2 Tracing the memoized fibonacci(4)

```
static Map<Integer, Long> memo = new HashMap<>();
static long fibonacci(int n) {
    if (memo.containsKey(n)) {
        return memo.get(n);
    }
    if (n == 0 || n == 1) {
        return n;
    }
    long result = fibonacci(n - 1) + fibonacci(n - 2);
    memo.put(n, result);
    return result;
}
```

Call	Argument	Event	Returns	`memo` (after)
1	4	not in memo, not a base case → `fibonacci(3)`	3	{}
2	3	not in memo → `fibonacci(2)`	2	{}
3	2	not in memo → `fibonacci(1)`	1	{}
4	1	base case	1	{}
5	0	base case; `fibonacci(2) = 1 + 0` is stored	0	{2=1}
6	1	base case; `fibonacci(3) = 1 + 1` is stored	1	{2=1, 3=2}
7	2	**in memo**, returned without computation; `fibonacci(4) = 2 + 1` is stored	1	{2=1, 3=2, 4=3}

Table 3.5.17: Trace of the call `fibonacci(4)` with an empty memo.

The first six calls are the same as in the version without memoization (Table 3.5.17). The difference appears in the seventh call. The right term of `fibonacci(4)` is `fibonacci(2)`, and this value was written to memory at the end of the fifth call. The method finds it on its first line and immediately returns 1. The version without memoization would make two more calls below `fib(2)` here: `fib(1)` and `fib(0)`. The total number of calls drops from 9 to 7. Base-case values are not written to memory; they need no computation anyway.

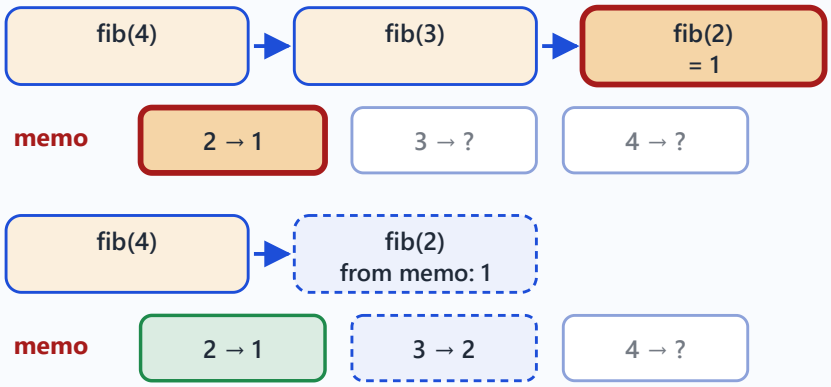


Figure 3.5.17: The call chain and the memo. Top: `fib(2) = 1` has been computed and `2 → 1` has been written to the memo. Bottom: the second `fib(2)` call reads its value from the memo (dashed box); the memo holds `2 → 1` and `3 → 2`.

3.5.8.3 The call tree and the gain

The effect of memoization becomes striking as n grows. Figure 3.5.18 shows the call tree of the memoized `fibonacci(5)`. The left arm is computed once from 5 down to 0. The `fib(2)` and `fib(3)` on the right come from the memo, and no new branches open below them. Nine calls are made instead of 15.

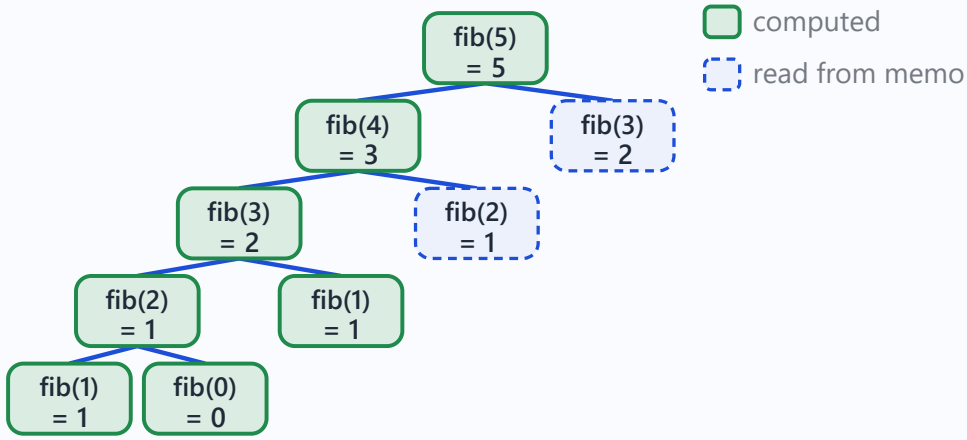


Figure 3.5.18: The call tree of the memoized `fibonacci(5)` : 9 calls. The dashed nodes are values read from the memo; the tree turns into a chain with short side branches.

In general, with an empty memo `fibonacci(n)` makes exactly $2n - 1$ calls for $n \geq 1$. While the left arm descends from `n` to 1, n calls are made. For each value from 2 to n the right term needs one more call; these are $n - 1$ calls. The total is $n + (n - 1) = 2n - 1$. Exponential growth has come down to linear. Table 3.5.18 compares the two versions with measurements.

n	Calls without memo	Calls with memo	Time without memo	Time with memo
5	15	9	< 0.1 ms	< 0.1 ms
20	21,891	39	≈ 0.3 ms	< 0.1 ms
40	331,160,281	79	≈ 0.5 s	< 1 ms
90	$\approx 9 \cdot 10^{18}$	179	centuries	< 1 ms

Table 3.5.18: Fibonacci without and with memoization (times measured with JDK 21).

The times vary from machine to machine; the ratio does not. The number of calls without memoization in the last row is 2F(91) − 1; even at a billion calls per second it would take about 300 years. The memoized version gives the same result in 179 calls.

This gain has a price: memory. The stored results hold as many entries as the value of n, and every entry contains an `Integer` and a `Long` object. Memoization spends memory to buy speed. In Fibonacci this trade is clearly profitable; in some problems, however, the number of results to be stored may be too large to fit in memory. Another limit is the stack: the left arm of the memoized version still descends n frames deep. `fibonacci(100000)` gives a `StackOverflowError` even with memoization, and its result would not fit in a `long` anyway.

3.5.8.4 Factorial with memoization

The same pattern can be applied to any recursive method: look in the memo first, otherwise compute, store, return.

```
static Map<Integer, Long> memo = new HashMap<>();

static long factorial(int n) {
    if (memo.containsKey(n)) {
        return memo.get(n);
    }
    if (n == 0 || n == 1) {
        return 1;
    }
    long result = n * factorial(n - 1);
    memo.put(n, result);
    return result;
}
```

Thanks to `long`, correct results are obtained up to 20!: `factorial(20)` = 2432902008176640000. `factorial(21)`, however, overflows and returns −4249290049419214848; the overflowed value is also written to the memo.

There is an interesting situation here. Factorial is not tree recursion; in a single `factorial(n)` call no value is computed twice. For this reason the memo does not speed up a single call; because of the `Map` operations it even slows it down slightly. The gain appears when the program asks for factorials

many times. Because `memo` is static, it is preserved between calls. After `factorial(10)` has been called, the values from 1! to 10! are in memory. If `factorial(12)` is then requested, only two new multiplications are made, 12 × 11! and 11 × 10!; 10! is read from the memo. In computations that use factorials many times, such as binomial coefficients, this gain is meaningful.

Common mistake

Forgetting to consult the memo or to write the result. If the `memo.put` line is not written, the memo stays empty and the method behaves like the version without memoization. If the `containsKey` check is written after the base case and after the recursive call, the memo is of no use either. The correct order: consult the memo, base case, compute, store, return.

3.5.9 Common mistakes

Most errors in recursive code fall into a few patterns. Some appear at compile time and some at run time; the most insidious give no message at all. Table 3.5.19 collects these patterns with the real messages of JDK 21.

Mistake	Example	Symptom	Fix
Missing or unreachable base case	`sum(0)`, base case `n == 1`	`Exception in thread "main" java.lang.StackOverflowError`	Widen the base case: `n <= 0`
No progress	`return n * factorial(n);`	`StackOverflowError`	Shrink the argument: `n - 1`
Progress skips the base case	`if (n == 0) return 0; return f(n - 3);`, `f(7)`	`StackOverflowError` (7, 4, 1, −2, …)	A base case that tests a range, such as `n <= 0`
Missing `return`	A method ending in `if … else if`	`error: missing return statement`	Make the last branch an `else`
Ignoring the result of the call	`factorial(n - 1);` on its own	No message; the value is lost	`return n * factorial(n - 1);`
Overflow	`int` factorial, Bell coefficient	No message; wrong result at 13!	`long` or `BigInteger`
Needless tree recursion	`fibonacci(50)` without memoization	A run lasting minutes	A loop or memoization

Table 3.5.19: Common mistakes in recursion.

The missing `return` error comes from the Java compiler not evaluating conditions. The following method is written to find the largest digit of a number:

```
static int maxDigit(int n) {
    if (n < 10) {
        return n;
    } else if (n % 10 > maxDigit(n / 10)) {
        return n % 10;
    } else if (n % 10 <= maxDigit(n / 10)) {
        return maxDigit(n / 10);
    }
}
```

```
error: missing return statement
    }
    ^
```

The two `else if` conditions are each other's opposite and one of them is certainly true. But the compiler does not see this; it does not compute the values of the conditions at compile time. What it sees is that all three conditions could be false, in which case the method would end without returning a value. When the last branch is made an `else`, the error disappears. The same method has a second problem: `maxDigit(n / 10)` may be computed three times in one call. This triples the number of calls at every digit. The correct way is to compute the subproblem's result once and keep it in a variable: `int sub = maxDigit(n / 10); return Math.max(n % 10, sub);`. This version returns 9 for 48729 and makes five calls.

Ignoring the result of a call is seen especially among beginners. The line `factorial(n - 1);` is a valid Java statement: the method is called, the value is computed and then discarded. The compiler gives no warning for this. The value returned by a recursive call must either be used inside a `return` expression or be written to a variable.

Overflow and needless tree recursion are the two most insidious errors, because neither gives a message. The program runs and produces a result. Overflow gives a wrong number. Tree recursion gives the right result very late. The way to prevent both is to think, while writing the method, about how large the input can grow and to test with boundary values.

3.5.10 Summary

- **Recursion:** a method calls itself with a smaller input. Java needs no special keyword; the method body calls its own name.
- **Base case and progress:** the base case is the smallest case, solved directly, and it stops the recursion. Every call must approach the base case (`n - 1`, `n / 10`, `substring`, `x % y`). If either of the two rules is broken, a `StackOverflowError` results.

- **Call stack:** a frame is opened for every call; a frame has its own parameters and local variables. Frames close in LIFO order. With the default stack the depth cannot exceed a few tens of thousands of calls; the limit varies with the JVM settings.
- **Descent and ascent:** work done before the call runs on the way down, work done after it on the way back. A method that prints on the way back reverses the order (`write2` , `toBinary`).
- **Sequences:** a recurrence relation translates directly into a recursive method. Fibonacci and Lucas call themselves twice; Catalan and Bell make their calls inside a loop. An `int` is enough up to 12!, a `long` up to 20!; overflow is silent.
- **Examples:** digit sum shrinks digit by digit, the palindrome test from both ends, and Euclid's algorithm by the remainder. The Towers of Hanoi makes $2^n - 1$ moves and takes a few lines with recursion.
- **Tree recursion:** multiple calls (Fibonacci, Hanoi) form a tree; when there is repetition the number of calls grows exponentially. `fibonacci(40)` without memoization makes 331,160,281 calls.
- **Iterative solution:** a loop with an accumulator variable runs in a single frame with constant memory and is usually faster. Every recursive solution can be written with a loop.
- **Tail recursion:** the last action is the call itself and the result is carried in an accumulator parameter. The accumulator's initial value is the identity element and is hidden by a wrapper method. Java does not perform tail call optimization.
- **Memoization:** computed results are stored in a `Map` and reused. Memoized Fibonacci makes 2n − 1 calls; its price is memory.

3.5.11 Key terms

recursion base case recursive step call stack stack frame stack overflow stack trace

tree recursion recurrence relation leap of faith iterative solution tail recursion accumulator

tail call optimization memoization Euclidean algorithm palindrome

---

3.5.12 Test Yourself

1. For the following method, what is `f(6)` ? How many times is the method called (including the first call)? What does the method compute?

```
static int f(int n) {
    if (n <= 0) return 0;
    return (n % 2 == 0 ? n : 0) + f(n - 1);
}
```

Answer

`f(6)` = 6 + 0 + 4 + 0 + 2 + 0 + 0 = 12. The arguments are 6, 5, 4, 3, 2, 1, 0; 7 calls are made. The method computes the sum of the even numbers from 1 to n. Because the base case is `n <= 0`, it also stops for negative input.

2. What does `g(3)` print?

```
static void g(int n) {
    if (n == 0) return;
    System.out.print(n);
    g(n - 1);
    System.out.print(n);
}
```

Answer

`321123`. The first `print` runs on the way down and writes 3, 2, 1. The second `print` runs on the way back; because the deepest frame returns first, 1, 2, 3 is written.

3. What value does the following exponentiation method return for `power(2, 10)`, and how many calls does it make? How many calls would a version that decreases the exponent by one, `base * power(base, exp - 1)`, make?

```
static long power(long base, int exp) {
    if (exp == 0) return 1;
    long y = power(base, exp / 2);
    return exp % 2 == 0 ? y * y : y * y * base;
}
```

Answer

It returns 1024; since the exponents are 10, 5, 2, 1, 0, 5 calls are made. `power(2, 5)` = 32 (odd exponent: 4 · 4 · 2), then 32 · 32 = 1024. The version that decreases the exponent by one would make 11 calls. Halving the exponent reduces the number of calls from O(e) to O(log e). Keeping the subresult in `y` is important; had `power(base, exp / 2) * power(base, exp / 2)` been written, the gain would have been lost.

4. How would the `countDigits` method be written as a tail-recursive method `countDigitsAcc(int`

`n, int k)` that uses an accumulator? What should `countDigitsAcc(48729, 0)` return?

Answer

```
static int countDigitsAcc(int n, int k) {
    if (n < 10) return k + 1;
    return countDigitsAcc(n / 10, k + 1);
}
```

`k` counts the digits dropped so far and starts at 0. (48729, 0) → (4872, 1) → (487, 2) → (48, 3) → (4, 4) → 4 + 1 = 5. Because its last action is the call, it is tail-recursive.

5. For `static int h(int n) { if (n == 0) return 1; return h(n - 1) + h(n - 1); }`, what is `h(4)` and how many calls are made? What happens to the number of calls if the last line is changed to `return 2 * h(n - 1);`?

Answer

`h(4)` = 2^4 = 16. Since every call opens two calls, the tree has 1 + 2 + 4 + 8 + 16 = 31 calls. With `2 * h(n - 1)` the same result is found in 5 calls. Calling the same subproblem twice raises the cost to exponential without changing the result.

6. How many calls does `gcd(89, 55)` make and what is its result? How many frames are on the stack at the deepest moment, counting `main`?

Answer

The GCD is 1. The chain is (89, 55) → (55, 34) → (34, 21) → (21, 13) → (13, 8) → (8, 5) → (5, 3) → (3, 2) → (2, 1) → (1, 0); 10 calls are made. All of them are on the stack at the same time, because Java does not optimize tail calls: 11 frames with `main`. Consecutive Fibonacci numbers are the case in which the remainder shrinks most slowly at each step.

7. What does the following method return for `count("kelebek", 'e')`? How many calls are made? Which is the base case?

```
static int count(String s, char c) {
    if (s.length() == 0) return 0;
    int k = s.charAt(0) == c ? 1 : 0;
    return k + count(s.substring(1), c);
}
```

Answer

It returns 3: "kelebek" contains three 'e's. The string gets one character shorter on every call; 7 calls for the seven letters and 1 call for the empty string make 8 calls in total. The base case is the empty string, which returns 0.

8. The Towers of Hanoi with 5 disks is solved with `solve(5, 'A', 'C', 'B')`. How many moves are printed and what is the first move? On which move is the largest disk moved?

Answer

$2^5 - 1 = 31$ moves. Because the number of disks is odd, the first move is `1: A->C`. The largest disk is moved from A to C on the 16th move, after the top 4 disks have been moved to B in $2^4 - 1 = 15$ moves.

3.5.13 Further reading

- A. B. Downey and C. Mayfield, *Think Java* (2nd ed., 2019), Chapter 8: recursive methods are explained with stack diagrams and the "leap of faith". The book can be read free online.
- R. Sedgewick and K. Wayne, *Computer Science: An Interdisciplinary Approach*, Section 2.3: the power of recursion through Euclid, Hanoi, Gray codes and recursive drawings; the exponential Fibonacci trap and its remedy with dynamic programming.
- P. Deitel and H. Deitel, *Java: How to Program*, Chapter 18: compares recursion with iteration and gives further examples such as fractal drawings, eight queens and mazes.
- H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs* (2nd ed., MIT Press, 1996), Section 1.2: the classic treatment of linear and tree recursion, iterative processes and the notion of tail calls.

3.5.14 References

1. P. Deitel, H. Deitel, *Java: How to Program (Early Objects)*, 11th ed., Pearson, 2017 (Chapter 18: Recursion).
2. A. B. Downey, C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019 (Chapter 8: Recursive Methods).
3. J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, G. Bierman, *The Java Language Specification, Java SE 21 Edition*, Oracle, 2023 (15.12: Method Invocation Expressions).
4. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson, 2022.
5. Oracle, *The Java Tutorials: Defining Methods; Returning a Value from a Method*, docs.oracle.com/javase/tutorial/java/javaOO/.
6. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016 (2.3: Recursion).

7. H. Abelson, G. J. Sussman, J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed., MIT Press, 1996.

PART II

Object-Oriented Programming

Chapters 4–5

PART II · CHAPTER 4

Objects and Classes, Inheritance and Polymorphism

Object-oriented programming brings data and the behavior that operates on it together in classes. The chapter starts with classes and objects, constructors, static members, the this reference, records, enum types and inner classes, and continues with encapsulation, access modifiers, immutable classes and packages. Inheritance is treated through super, overriding and the Object class, and polymorphism through reference types, casting, dynamic binding and overloading. The final part covers abstract classes, interfaces, lambda expressions, sealed types, generics, annotations and widely used design patterns.

NOTES IN THIS CHAPTER

4.1 Objects and Classes

4.2 Encapsulation

4.3 Inheritance

4.4 Polymorphism

4.5 Abstract Classes and Interfaces

PART II · CHAPTER 4.1

Objects and Classes

The programs so far consisted of `static` methods and the variables passed to them. The data lived in one place and the code that processed it in another. Object-oriented programming brings the two together: a student's name, number and grade average live in the same unit as the methods that work on them. The blueprint for such a unit is a **class**; each concrete instance produced from the blueprint is an **object**. These notes show step by step how a class is written, what happens in memory when an object is created with `new`, and why a variable holds a reference to an object rather than the object itself. Instance and `static` members, initialization order, `this` and constructor chaining, the difference between `==` and `equals`, wrapper classes, records, `enum` and inner classes are also covered. The notes end by designing a fraction class from scratch. All programs were compiled and run with JDK 21; the outputs, the final values in the trace tables and the compiler messages come from those runs.

LEARNING OUTCOMES

Class and object
Explain a class as a blueprint and an object as a concrete instance made from it; write a class with fields and methods.

Creating objects
Use the `new` operator and constructors; know the default values of fields and the default constructor the compiler adds.

Instance and class members
Distinguish instance members from `static` members; trace the order in which initializer blocks and constructors run.

References
Show with a memory diagram that a variable holds a reference, not the object; explain the difference between `this`, `==` and `equals`.

Constructor chaining and access
Chain constructors with `this(...)`; explain the purpose of access modifiers and data hiding.

Special kinds of classes
Recognize wrapper classes, records, the `enum` type and inner classes; design a class from scratch with the `Rational` example.

4.1.1 Object-oriented programming

Imagine writing a student information system with `static` methods. Each student's name is kept in one array, the number in a second array and the grade average in a third. Methods such as `computeAverage`, `registerCourse` and `printTranscript` take these arrays as parameters. As the program grows, problems appear. Because the arrays are separate, only the index links the name of student 5 to the average of student 5. If one array is sorted and the other forgotten, names end up paired with other students' grades. Adding a new piece of information to a student, say the department, means a fourth array and new parameters to carry it. Every method that touches these arrays has to change.

Object-oriented programming answers this scattering with a different organization. All the information about a student is gathered in a single unit. The code that works on this information is written inside the same unit. The program is built from such units asking one another to do work.

Definition 4.1.1

Object-oriented programming (OOP) is a programming approach that builds a program from interacting objects. Each object carries its own data and offers operations that work on that data. Objects cooperate by calling one another's methods.

4.1.1.1 Object: state and behavior

Every clearly distinguishable entity in the real world can be modeled as an object. A student, a desk, a circle, a button on the screen, even a loan taken from a bank is an object. Each of these objects has two things. The first is the object's **state**: the data it holds at that moment. A student's name, number and department make up the student's state. The second is the object's **behavior**: the things the object can do. A student can enroll in a course and view the grades. Table 4.1.1 puts the state and behavior of five objects side by side.

Object	Fields (state)	Behaviors
Student	name, id, department	`enroll()`, `viewGrades()`
Desk	length, width, material	`move()`
Circle	center, radius	`area()`, `perimeter()`
Button	label, color, enabled	`click()`

Object	Fields (state)	Behaviors
Loan	amount, interest, term	`computeInstallment()`

Table 4.1.1: Objects from the real world. State is represented by fields, behavior by methods.

In a programming language the counterparts of state and behavior are clear. The pieces of data an object stores are called **fields**, and the things it can do are called **methods**. Objects get work done by calling one another's methods. In a school program, student, teacher and course objects "send messages" to one another in this way. A teacher object calls the `enterGrade` method of a course object; the course object then updates the student's record. Sending a message means calling an object's method in Java.

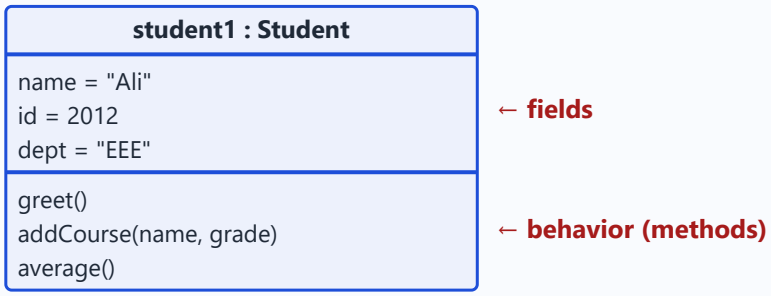


Figure 4.1.1: The `student1` object. The upper part holds the fields and their current values, the lower part the object's methods. The data and the code that uses it are inside the same box.

Figure 4.1.1 draws the object `student1` of type `Student`. The upper part of the box contains the fields `name`, `id` and `dept` with their values, and the lower part contains the methods `greet()`, `addCourse(name, grade)` and `average()`. The `average()` method takes no array parameter; the grades it needs are already inside the object. This togetherness of data and code makes the program better organized. The code becomes reusable, and because it resembles its real-world counterpart it is easier to understand.

4.1.1.2 Procedural and object-oriented approaches

The programs so far were written in the **procedural** style. In this style the basic unit is the function. Data sits in separate variables and is passed as a parameter to every function that wants it. In the object-oriented style the basic unit is the object; data is stored inside the object and is exposed to the outside only through the paths the object allows. Table 4.1.2 compares the two approaches from five angles.

	Procedural approach	Object-oriented approach
Basic unit	Function (method)	Object (data + methods)
Data	In separate variables; every function can reach it	Inside the object; exposed to the outside in a controlled way

	Procedural approach	Object-oriented approach
Example	`area(length, width)`	`rect.area()`
Large programs	A change affects many functions	A change mostly stays inside one class
Reuse	By copying functions	By using classes and inheritance

Table 4.1.2: Comparison of the procedural and object-oriented approaches.

The "Example" row of the table shows the difference in the shortest way. In the procedural call the `area` method expects the length and width of the rectangle as parameters. The caller has to store these two numbers somewhere and pass them in the right order. In the object-oriented call the length and width are inside the rectangle object. The caller only says whose area it wants: `rect.area()`.

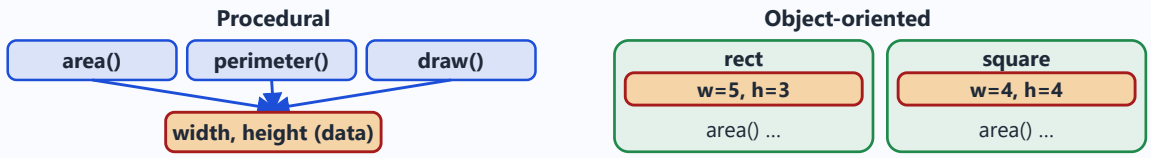


Figure 4.1.2: Left, the procedural layout: three functions reach the same shared data from outside. Right, the object-oriented layout: each object carries its own data (`length`, `width`) together with the methods that work on it.

The "Large programs" row states the real gain. On the left of Figure 4.1.2 three functions depend on the same data. If the form of the data changes, say corner coordinates are stored instead of length and width, all three functions must change. In the layout on the right the data is closed inside a class. A change of form affects the methods of that class; the code that uses the class goes on writing the `area()` call unchanged.

In Java all code is written inside a class. So the programs so far did have a class, but those classes were only containers holding `static` methods together; no object was ever made from them. In object-oriented programming classes do their real job: they produce objects.

4.1.1.3 Three basic features

Three features set object-oriented languages apart from procedural ones. The subject of these notes is the class and the object themselves. Still, knowing from the start what the three features are for makes it easier to understand the decisions made when writing a class.

- **Encapsulation.** An object can hide the details of its fields and behavior from other objects. To the outside it is like a closed box, used only through the methods it permits. The circuits inside a television are in a closed case; the user reaches only the buttons of the remote control.
- **Inheritance.** A class can take over the fields and methods of another class. In this way common code is written once and classes are organized in a hierarchy.

- **Polymorphism.** An object can be seen as more than one type. The same call produces different behavior on different objects. The call `shape.area()` computes πr^2 for a circle and a^2 for a square.

A fourth feature, **abstraction**, is often added to this trio. Abstraction means hiding unnecessary detail and showing only the important features. A `BankAccount` class models not all the properties of a real account but those useful to the program, for example the owner's name and the balance. The color of the door of the branch where the account was opened does not enter the model.

4.1.2 Class and object

A factory produces thousands of cars of the same model. Every car has a color, a chassis number and an odometer, and these values vary from car to car. But all of them come from the same drawing: the drawing determines which parts there will be and how the engine will work. In Java the counterpart of this drawing is the class, and the counterpart of each car produced is the object.

4.1.2.1 Class: the blueprint of an object

Definition 4.1.2

A **class** is the blueprint of an object. It defines the data (fields) the objects will carry and the operations (methods) they can perform. Any number of objects carrying different data can be produced from one class.

Definition 4.1.3

An **object** is an **instance** of a class: the concrete form, created in memory, of the structure the class describes. Each object carries its own copy of the fields the class defines.

The relationship between class and object is well explained by a cookie cutter. The cutter is made once and determines the shape of the cookie. Each cookie cut with it is a separate cookie: one may get chocolate, another hazelnut. The cutter itself is not eaten. A class likewise carries no data by itself; the objects produced from it carry the data.

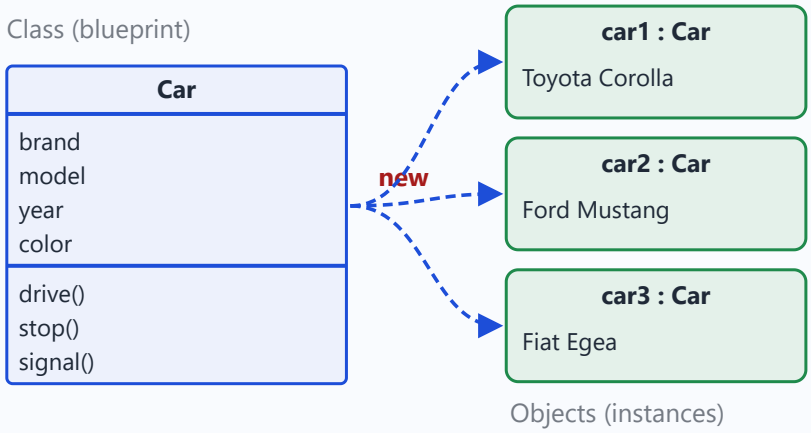


Figure 4.1.3: The `Car` class and three objects produced from it with `new`. The class defines the field names and methods; each object puts its own values in these fields.

On the left of Figure 4.1.3 the dashed box is the `Car` class. The class defines the fields `brand`, `model`, `year` and `color` and the methods `drive()`, `stop()` and `signal()`. The three objects on the right are produced from the same class: a Toyota Corolla, a Ford Mustang and a Fiat Egea. All three have the same field names, but the values differ. Two objects produced from the same class carry the same field names; the field values of each belong to itself. Changing the color of one does not change the color of the other.

Classes are usually drawn with a **UML class diagram** (Unified Modeling Language). In the diagram a class is a box with three compartments: the name of the class at the top, the fields in the middle, the methods at the bottom. The sign before each member shows its access level: `-` means `private` and `+` means `public`. Types are written next to fields and return types next to methods. Objects are drawn with a similar box, but the header holds the object's name and class in the form `car1 : Car`, and the header is underlined.

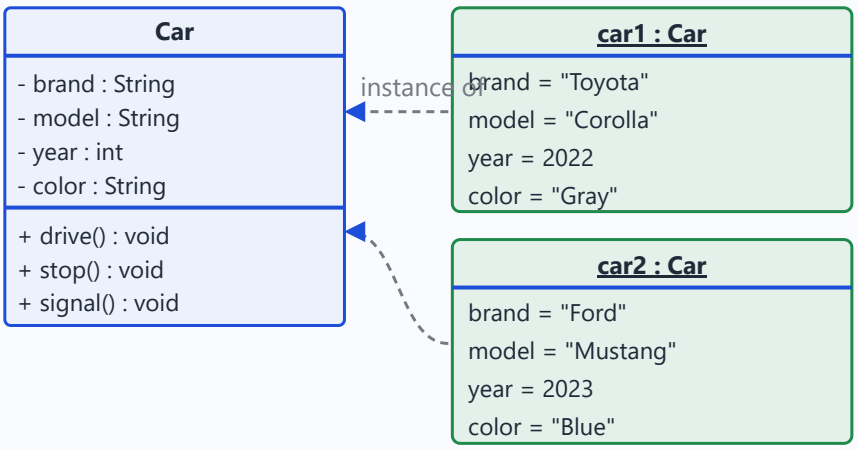


Figure 4.1.4: UML notation. On the left the `Car` class (name / fields / methods), on the right two objects of this class. The object boxes show the current values of the fields; the dashed arrows show the "instance of"

relation.

4.1.2.2 Defining a class

The code below is the first Java version of the `Car` class in Figure 4.1.4. In the diagram the fields are marked `private`; in this first version no access modifier is written. The fields therefore have package access, and another class in the same folder can assign values to them directly.

```
public class Car {
  // Fields
  String brand;
  String model;
  int year;
  String color;

  // Methods
  void drive() {
    System.out.println(brand + " is moving.");
  }

  void stop() {
    System.out.println(brand + " stopped.");
  }

  void signal() {
    System.out.println(brand + " is signaling.");
  }
}
```

A class definition begins with the `class` keyword and the name of the class. The body of the class is between curly braces. The name starts with a capital letter, and if it consists of several words the first letter of each word is capitalized: `Car`, `BankAccount`, `Rational`. A `public` class must be in a file with the same name as the class. The definition `public class Car` is written in the file `Car.java`. A file may contain several classes, but at most one of them can be `public`.

Fields are declared in the body of the class, outside the methods. The declaration resembles a local variable declaration: a type and a name. The line `String brand;` says that every `Car` object will have a `String` field called `brand`. Fields can be used directly from every method of the class. The `drive()` method does not take `brand` as a parameter; it recognizes it as a field of the class.

There is no `static` in the headers of the methods. All the methods so far were `static` and were called with the class name, without an object. The methods here, in contrast, are methods of a particular car. When `drive()` is called, the object through which the method is called determines whose brand is printed. Such methods are called **instance methods**.

The class has no `main` method. So `Car` cannot be run on its own; it is a building block that another class uses. Most classes in real programs are like this: only one class carries `main`, and the others define the types it uses.

4.1.2.3 Using a class

The `CarTest` class creates two `Car` objects, gives values to their fields and calls their methods. If the two files are in the same folder, they are compiled with `javac Car.java CarTest.java` and run with `java CarTest`.

```
public class CarTest {
  public static void main(String[] args) {
    Car car1 = new Car();
    car1.brand = "Toyota";
    car1.model = "Corolla";
    car1.year = 2022;
    car1.color = "Gray";

    Car car2 = new Car();
    car2.brand = "Ford";
    car2.model = "Mustang";
    car2.year = 2023;
    car2.color = "Blue";

    car1.drive();
    car2.signal();
    car1.stop();
  }
}
```

```
Toyota is moving.
Ford is signaling.
Toyota stopped.
```

The `new Car()` in the first line creates a new `Car` object in memory. The declaration `Car car1` opens a variable of type `Car`. After the assignment `car1` points to the newly created object. The type of the variable is a class name; in Java every class is also a new type.

The next four lines assign values to the fields of the object. The dot in `car1.brand = "Toyota";` means "the `brand` field of the object that `car1` points to". This sign is called the **dot operator**. For the second object other values are given to the same fields. Now there are two separate `brand` fields in memory: one holds `"Toyota"` and the other `"Ford"`.

The last three lines call methods. In the call `car1.drive()`, the `brand` in the body of the method is the brand of `car1`: `Toyota is moving.` is printed. In the call `car2.signal()` the same name denotes the brand of `car2`: `Ford is signaling.` The method code is written once; the object to the left of the dot decides whose data it works with.

Assigning values to fields directly from outside like this is not a good habit. The line `car1.year = -5;` also compiles and puts the object in a meaningless state. The right way to give the fields their initial values is constructors; the right way to change values later is methods that perform checks. The style here is only to introduce the dot operator.

4.1.2.4 Members of a class

Everything defined in the body of a class is called a **member** of the class. There are two basic kinds of members:

- **Fields** represent the data objects can carry. In a `Student` class, `name` and `surname` are fields.
- **Methods** define the operations objects can perform. In a `Dog` class, `bark()` and `run()` are methods.

Every member has two further properties. The first is the **access level**: a member can be `private`, `protected`, `public`, or, when none of these is written, package-private. The access level determines who may use the member. The second is **ownership**: a member either exists separately in every object (instance member, the default) or is declared `static` and belongs to the class itself.

Two more kinds of constructs can appear in the body of a class. **Constructors** are written inside the class and resemble methods, but they run only when an object is created. The Java language specification counts them not as methods but as a separate kind of member. Other classes can also be defined inside a class; these are called **inner classes**.

4.1.2.5 Access modifiers and data hiding

> **Definition 4.1.4**
>
> An **access modifier** is a keyword that specifies who can reach a member from outside the class. Java has four access levels: `public` (everyone), `protected` (the same package and subclasses), **package access** when nothing is written (classes in the same package) and `private` (only the class itself).

The general rule is simple: fields are declared `private` and the methods offered to the outside are declared `public`. The value of a field is read from outside with a **getter**; its name traditionally starts with `get`. The value is changed with a **setter**; its name starts with `set`. The real job of a setter is to check the incoming value and reject invalid ones.

```
public class BankAccount {
  private double balance;   // hidden field
  public double getBalance() {
    return balance;
  }
  public void setBalance(double b) {
    if (b >= 0) balance = b;  // reject invalid
  }
}
// in another class:
acct.balance = -500;
```

The `balance` field is `private`. A class outside `BankAccount` can neither read nor change this field. When the last line of the code is written in another class, the compiler gives this message:

```
class BankAccount {
  private double balance;   // hidden field
}
public class PrivateAccess {
  public static void main(String[] args) {
    BankAccount acct = new BankAccount();
    acct.balance = -500;
  }
}
```

```
PrivateAccess.java:7: error: balance has private access in BankAccount
    acct.balance = -500;
         ^
1 error
```

The only way in from outside is the two `public` methods. `getBalance()` returns the value. `setBalance(b)` checks the new value before writing it: if `b` is negative, no assignment is made. The test program below first tries to give a valid and then an invalid balance:

```
public class AccountTest {
  public static void main(String[] args) {
    BankAccount acct = new BankAccount();
    acct.setBalance(1500);
    acct.setBalance(-500);   // rejected
    System.out.println(acct.getBalance());
  }
}
```

```
1500.0
```

Because `-500` is rejected, the balance stays `1500.0`. The program cannot make the balance negative; the rule that prevents this is written in a single place, inside `setBalance`. Had the field been `public`, every line that changes the field would have had to check the same rule separately, and forgetting just one would corrupt the object.

Definition 4.1.5

Data hiding is the arrangement in which only the methods of the class itself can access the fields directly. Fields are made `private`, and the outside world talks to the object only through `public` methods. Data hiding is the foundation of encapsulation.

The second gain of data hiding is freedom to change. Suppose it is decided to keep the balance in a `long` field in cents instead of a `double`. Because the field is `private`, no line outside uses it. Only the bodies of two methods change; the code that uses the class does not change at all.

Common mistake

Adding a get and a set method to every `private` field without thinking. A method such as `public void setBalance(double b) { balance = b; }` that performs no check effectively makes the field `public`. A setter is written only if the field really has to be changed from outside, and it rejects invalid values. A field that must never change has no setter.

4.1.3 Constructors and object creation

In `CarTest` each object was set up in five lines: first `new Car()`, then four separate assignments. In this arrangement, if one of the assignments is forgotten, the object is left half-built, and the error shows up only at run time, perhaps much later. Constructors make sure an object is put into a valid state at the moment it is created.

4.1.3.1 What is a constructor?

Definition 4.1.6

A **constructor** is a special block of code that runs while an object is being created. It is called automatically whenever an object is created with `new`. Its name is exactly the same as the class name (including case) and it has no return type; not even `void` is written.

The job of a constructor is to give the fields their initial values and to do whatever else is needed to make the object ready for use. The constructor of the `Student` class below takes two parameters and writes them to the fields of the object.

```
public class Student {
  private String name;
  private int id;

  // Constructor: class name, no return type
  public Student(String name, int id) {
    this.name = name;
    this.id = id;
  }
}

// Usage:
Student s = new Student("Ali", 2012);
```

The header of a constructor resembles a method header, but with two differences. The name is the name of the class: `Student`. There is no return type before the name. A constructor does not return a value; the result of a `new` expression is always the new object itself. The line `this.name = name;` in the body writes the value of the `name` parameter to the `name` field of the object. Here `this` means "the object being created".

Constructors are usually declared `public`, because the code that creates the object is outside the class. Sometimes one wants to restrict object creation. In that case the constructor is made `private` and `new` cannot be called from outside the class. Singleton classes are built this way. A class may have several constructors with different parameter lists.

4.1.3.2 The new expression

The general form of object creation is this:

```
ClassName objectName = new ClassName(arguments);

Car corolla = new Car("Toyota", "Corolla", 2022);
Scanner in = new Scanner(System.in);
Book book = new Book();
```

The first line shows the pattern and the next three lines show three examples. To the left of the equals sign is a variable declaration and to the right a `new` expression. Java's own classes are used the same way: to read from the keyboard one writes `new Scanner(System.in)`, and for a modifiable text `new StringBuilder("Java")`. The program below creates and uses four objects.

```
import java.util.Scanner;
public class CreateDemo {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    StringBuilder text = new StringBuilder("Java");
    String title = "Objects";           // shortcut
    String author = new String("Sercan");  // long way
    text.append(" 21");
    System.out.println(text + " / " + title);
    System.out.println(author.length());
    System.out.println(in.nextInt() * 2);
  }
}
```

When `21` is typed at the keyboard, the program prints:

```
Java 21 / Objects
6
42
```

The `new` expression works in three steps:

1. `new` reserves space for the object in the region of memory called the **heap**. All the fields of the object first receive their default values: numbers become 0, `boolean` fields `false`, reference fields `null`.
2. The constructor matching the arguments is called and gives the fields their initial values.
3. The result of the `new` expression is the **reference** to the new object. This reference is assigned to the variable to the left of the equals sign.

With `String` there is a familiar exception to this rule: writing a text literal also yields a `String` object. The line `String title = "Objects";` puts the reference of a `String` object into `title` without using `new`. Writing `new String("Sercan")` is also valid, but it needlessly produces a new object each time it runs. Writing just `String title;` creates no object at all; it only declares a variable that can hold a reference.

The `6` in the second line of the output is the result of the call `author.length()`: `"Sercan"` has six letters. This call is also an instance method call made with the dot operator. There is no difference in usage between the classes a programmer writes and Java's ready-made classes.

4.1.3.3 References

An object cannot be touched directly. A program works with an object only through the **reference** that leads to it. A variable of a class type stores not the object itself but this reference.

Definition 4.1.7

A **reference** is a value that points to the location of an object on the heap. It can be thought of as the object's address in memory. Variables of class types (`Car` , `String` , `Scanner` , arrays) hold references. A reference that points to no object has the value `null` .

Variables of primitive types such as `int` , `double` and `boolean` hold the value themselves. After the line `int x = 5;` the number 5 sits in the box of `x` . After the line `Car a = new Car(...);` , however, the box of `a` holds not a car but an arrow pointing to where the car is. An easy way to remember this difference is an analogy: the variable is a television remote control and the object is the television. Copying the remote does not create a second television; two remotes switch the same television on and off.

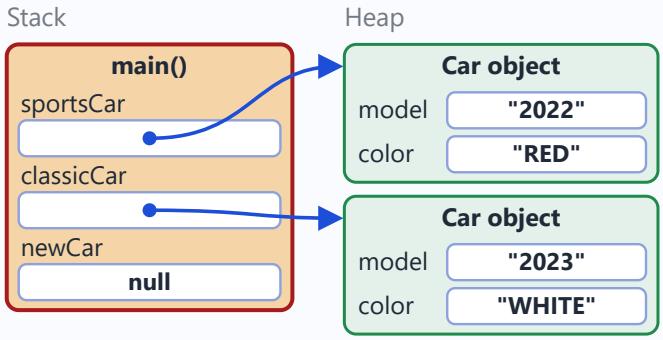


Figure 4.1.5: Local variables are on the stack and objects on the heap. `sportsCar` and `classicCar` point to two separate objects; `newCar` does not point to any object yet (`null`).

Memory is divided into two regions. The frames and local variables of running methods are on the **stack**. Objects are kept on the **heap**. In Figure 4.1.5 the frame of `main` is on the left. The arrows leaving the boxes `sportsCar` and `classicCar` in the frame reach the two objects on the heap. The third variable, `newCar` , has been declared but no object has been assigned to it yet; its value is `null` . This drawing convention is used throughout the rest of the notes: a box is a variable, an arrow is a reference, and a framed box on the heap is an object.

4.1.3.4 Tracing object creation step by step

The `Car` class below contains two fields, a constructor with two parameters and a method. `main` creates two objects and prints the information of both.

```
public class Car {
  private String model;
  private String color;

  public Car(String model, String color) {
    this.model = model;
    this.color = color;
  }

  public void printInfo() {
    System.out.println(model + " - " + color);
  }

  public static void main(String[] args) {
    Car sportsCar = new Car("2022", "RED");
    Car classicCar = new Car("2023", "WHITE");
    sportsCar.printInfo();
    classicCar.printInfo();
  }
}
```

```
2022 - RED
2023 - WHITE
```

Table 4.1.3 gives the eight steps of the program. The line numbers are the lines of the code above. The "Variables" column shows the variables of the topmost frame at that moment; the notation `→#1` means the variable points to object number 1 on the heap, and `?` means the variable has not been given a value yet.

Step	Line	Call stack	Variables	New output
1	15: `Car sportsCar = new Car("2022", "RED");`	main()	sportsCar=?, classicCar=?	—
2	5: `public Car(String model, String color) {`	main() → Car(model, color)	model="2022", color="RED"	—
3	6: `this.model = model;`	main() → Car(model, color)	model="2022", color="RED"	—
4	7: `this.color = color;`	main() → Car(model, color)	model="2022", color="RED"	—
5	15: `Car sportsCar = new Car("2022", "RED");`	main()	sportsCar=→#1, classicCar=?	—
6	16: `Car classicCar = new Car("2023", "WHITE");`	main()	sportsCar=→#1, classicCar=→#2	—
7	17: `sportsCar.printInfo();` 11: `System.out.println(model + " - " + color);`	main() → printInfo()	sportsCar=→#1, classicCar=→#2	`2022 - RED`

Step	Line	Call stack	Variables	New output
8	18: `classicCar.printInfo();` 11: `System.out.println(model + " - " + color);`	main() → printInfo()	sportsCar=→#1, classicCar=→#2	`2023 - WHITE`

Table 4.1.3: Trace of the calls `new Car(...)` and `printInfo()`.

In the first step line 15 starts to run. `new` reserves space for a `Car` object on the heap. `model` and `color` are fields of type `String`, that is, of a reference type; their default values are `null`. The object exists at this moment, but it is empty inside.

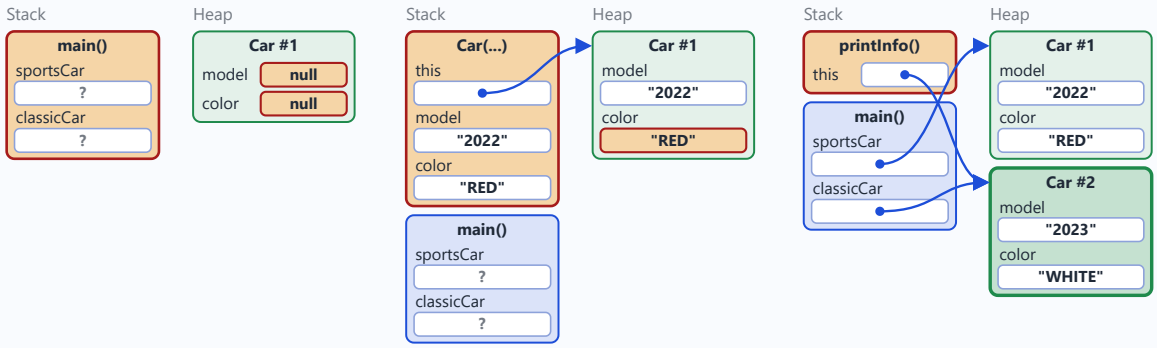


Figure 4.1.6: Three moments. Left: `new` has allocated the object and the fields are `null`. Middle: the constructor is running; `this` in the constructor's frame points to the new object and two fields have been filled. Right: the second `printInfo()` call; this time `this` points to object number 2.

In the second step the constructor is called and a new frame is opened on the stack. The parameters receive copies of the arguments: `model = "2022"`, `color = "RED"`. The frame has one more variable: `this`. When calling the constructor, Java quietly passes the reference of the new object under the name `this`. In the third and fourth steps `this.model = model;` and `this.color = color;` run. In each line `this.model` to the left of the equals sign is the field of the object, and `model` to the right is the parameter. After these two lines the object is ready (Figure 4.1.6, middle).

In the fifth step the constructor ends and its frame is removed. The parameters and `this` vanish with it, but the object stays on the heap. The reference, the value of the `new` expression, is assigned to `sportsCar`. The sixth step does the same for the second object. A separate object with its own copies of `model` and `color` appears on the heap, and `classicCar` points to it.

In the seventh step `sportsCar.printInfo()` is called. The method's frame also has a `this`, and its value is the reference to the left of the dot: object number 1. The names `model` and `color` in the body of the method mean `this.model` and `this.color`. That is why `2022 - RED` is printed. In the eighth step the same method is called through `classicCar`; this time `this` points to object number 2 and `2023 - WHITE` is printed. The code is single; the data are separate for each object.

The space an object occupies in memory is the sum of its fields. Methods are not copied into every object; the class's code exists in memory once. So creating a thousand `Car` objects takes a thousand times the space of two references, but the code of `printInfo` occupies space only once.

4.1.3.5 The default constructor and default values

The first `Car` class had no constructor at all, yet `new Car()` compiled without trouble. The reason is help from the compiler.

> **Definition 4.1.8**
>
> The **default constructor** is the parameterless constructor with an empty body that the compiler adds when no constructor has been written in the class. If even a single constructor is written in the class, the compiler does not add the default constructor.

A constructor is always called when an object is created; the default constructor makes sure that this call has a target when the programmer has not written one. Because its body is empty, it gives no values to the fields. Fields that are not given a value stay at the default values they received in the first step of `new`. Table 4.1.4 lists these values.

Field type	Default value
`byte`, `short`, `int`, `long`	`0`
`float`, `double`	`0.0`
`boolean`	`false`
`char`	`'\u0000'` (code 0)
Reference (`String`, arrays, objects)	`null`

Table 4.1.4: Default values of fields. Local variables get no default value.

```
class Point3 {
  int x;
  double weight;
  boolean visible;
  char letter;
  String name;
}
public class Defaults {
  public static void main(String[] args) {
    Point3 n = new Point3();   // default constructor
    System.out.println(n.x + " " + n.weight);
    System.out.println(n.visible + " " + n.name);
    System.out.println((int) n.letter);
  }
}
```

```
0 0.0
false null
0
```

The `Point3` class has no constructor; `new Point3()` calls the default constructor the compiler added. None of the five fields has been given a value, and each holds the value in the table. The value of the `char` field is the character with code 0. This character is not visible on the screen, so the program converts it to a number with `(int)` and prints it: `0`.

The default value rule is only for fields and array elements. A local variable inside a method gets no default value. The compiler does not accept a local variable that is read without being given a value:

```
public class Uninitialized {
  public static void main(String[] args) {
    StringBuilder sb;
    sb.append("x");
  }
}
```

```
Uninitialized.java:4: error: variable sb might not have been initialized
    sb.append("x");
    ^
1 error
```

The reason for this difference is safety. The compiler can examine all the paths inside a method and see whether a local variable receives a value before it is read. For fields it cannot do this, because a field can be read from any method in any order. Java therefore starts fields at least with a defined

value.

4.1.3.6 No-argument and parameterized constructors

More than one constructor can be written for a class. The `Vehicle` class below has two constructors, one without parameters and one with three.

```
public class Vehicle {
  private String brand;
  private String model;
  private int year;
  public Vehicle() {                // no-arg constructor
    brand = "Unknown";
    model = "Unknown";
    year = 0;
  }
  public Vehicle(String brand, String model, int year) {
    this.brand = brand;
    this.model = model;
    this.year = year;
  }
  public void printInfo() {
    System.out.println(brand + " " + model + " " + year);
  }
  public static void main(String[] args) {
    Vehicle a1 = new Vehicle();
    Vehicle a2 = new Vehicle("Renault", "Clio", 2021);
    a1.printInfo();
    a2.printInfo();
  }
}
```

```
Unknown Unknown 0
Renault Clio 2021
```

`Vehicle()` is a no-argument constructor written by the programmer. It differs from the default constructor in that it gives the fields meaningful initial values: brand and model become `"Unknown"` instead of `null`. The two constructors have the same name and are told apart by their parameter lists. This is the constructor counterpart of overloading in methods: **constructor overloading**. The arguments in the `new` expression decide which constructor runs. `new Vehicle()` calls the no-argument constructor and `new Vehicle("Renault", "Clio", 2021)` calls the three-parameter one.

Once any constructor has been written in a class, the compiler no longer adds a default constructor. In a class that has only a parameterized constructor, creating an object without arguments with `new` is a compile error:

```
public class VehicleDefaults {
  private String brand;
  public VehicleDefaults(String brand) {
    this.brand = brand;
  }
  public static void main(String[] args) {
    VehicleDefaults a = new VehicleDefaults();
  }
}
```

```
VehicleDefaults.java:7: error: constructor VehicleDefaults in class VehicleDefaults cannot
be applied to given types;
    VehicleDefaults a = new VehicleDefaults();
                        ^
  required: String
  found:    no arguments
  reason: actual and formal argument lists differ in length
1 error
```

The lower lines of the message say clearly what was expected: `required: String`, that is, a `String` argument is expected; `found: no arguments`, that is, no argument was given. If creating an object without arguments is needed, a no-argument constructor is also written by hand.

Common mistake

Writing a return type on a constructor. `public void CtorError(String m)` is not a constructor but an ordinary method that happens to have the same name as the class. Since the class has no other constructor, the compiler adds the no-argument default constructor, and `new CtorError("hello")` is rejected with this message:

```
CtorError.java:7: error: constructor CtorError in class CtorError cannot be applied
to given types;
    CtorError h = new CtorError("hello");
              ^
  required: no arguments
  found:    String
  reason: actual and formal argument lists differ in length
1 error
```

The message saying `required: no arguments` shows that the compiler sees only the no-argument constructor. When `void` is removed, the method becomes a constructor.

4.1.4 Fields

Fields are variables defined inside a class, outside the methods. They differ from local variables in three respects. They receive default values. They do not vanish when a method ends; they live as long as the object. They are visible from all the methods of the class. Whom a field belongs to depends on the word `static`.

4.1.4.1 Instance fields and class fields

Definition 4.1.9

An **instance field** is a field declared without `static`; every object has a separate copy of it. A **static field** (class field) is a field declared with `static`; the class has a single copy and all objects share it.

	Instance field	Class field (`static`)
Declaration	`private String name;`	`private static int accountCount;`
Number of copies	A separate copy in every object	A single copy in the class, shared by all objects
When created?	With every `new`	Once, when the class is loaded into memory
Access	`object.name`	`ClassName.accountCount`
Example	`Student` : `name`, `surname`	`BankAccount` : `accountCount`

Table 4.1.5: Instance fields compared with class fields.

Whether a field is an instance field or a class field is decided by this question: does the value vary from object to object? Every student's name is different; `name` is an instance field. The total number of accounts opened in the bank, on the other hand, is a single number. Keeping a separate counter in every account object would be both unnecessary and the counters would show different values. For this reason the counter is declared `static`.

4.1.4.2 Tracing a static counter

In the class below each account keeps its own owner's name, while all accounts share a single counter. Every time the constructor runs, the counter goes up by one.

```
public class BankAccount {
  private String owner;
  private static int accountCount = 0;

  public BankAccount(String owner) {
    this.owner = owner;
    accountCount++;
  }

  public static int getAccountCount() {
    return accountCount;
  }

  public static void main(String[] args) {
    BankAccount h1 = new BankAccount("Ali");
    BankAccount h2 = new BankAccount("Ayşe");
    BankAccount h3 = new BankAccount("Can");
    System.out.println(h2.owner);
    int count = BankAccount.getAccountCount();
    System.out.println(count);
  }
}
```

```
Ayşe
3
```

Step	Line	Call stack	Variables	New output
1	3: `private static int accountCount = 0;`	(class loading)	accountCount=0	—
2	5: `public BankAccount(String owner) {` 6: `this.owner = owner;` 7: `accountCount++;`	main() → BankAccount("Ali")	accountCount=1, h1=?	—
3	15: `BankAccount h1 = new BankAccount("Ali");`	main()	accountCount=1, h1=→#1	—
4	16: `BankAccount h2 = new BankAccount("Ayşe");`	main()	accountCount=2, h1=→#1, h2=→#2	—
5	17: `BankAccount h3 = new BankAccount("Can");`	main()	accountCount=3, h1=→#1, h2=→#2, h3=→#3	—
6	18: `System.out.println(h2.owner);`	main()	accountCount=3	`Ayşe`
7	19: `int count = BankAccount.getAccountCount();` 10: `public static int getAccountCount() {` 11: `return accountCount;`	main() → getAccountCount()	accountCount=3	—

Step	Line	Call stack	Variables	New output
8	20: `System.out.println(count);`	main()	accountCount=3, count=3	`3`

Table 4.1.6: Trace of the `static` counter. In the Variables column, `accountCount` is the value of the class field.

The first step happens before `main`. When the `BankAccount` class is loaded into memory, the `static` field is created once and receives its initial value: 0. At this moment there is no `BankAccount` object on the heap, but the counter exists. A `static` field is part of the class, not of the objects, and its life begins when the class is loaded.

In the second step `new BankAccount("Ali")` creates the first object. The first line of the constructor writes `"Ali"` to the object's own `owner` field. The `accountCount++` in the second line raises the shared counter to 1. From inside the constructor the `static` field can be reached without writing the class name; the compiler knows the name is a class field. In the third step `h1` starts pointing to this object.

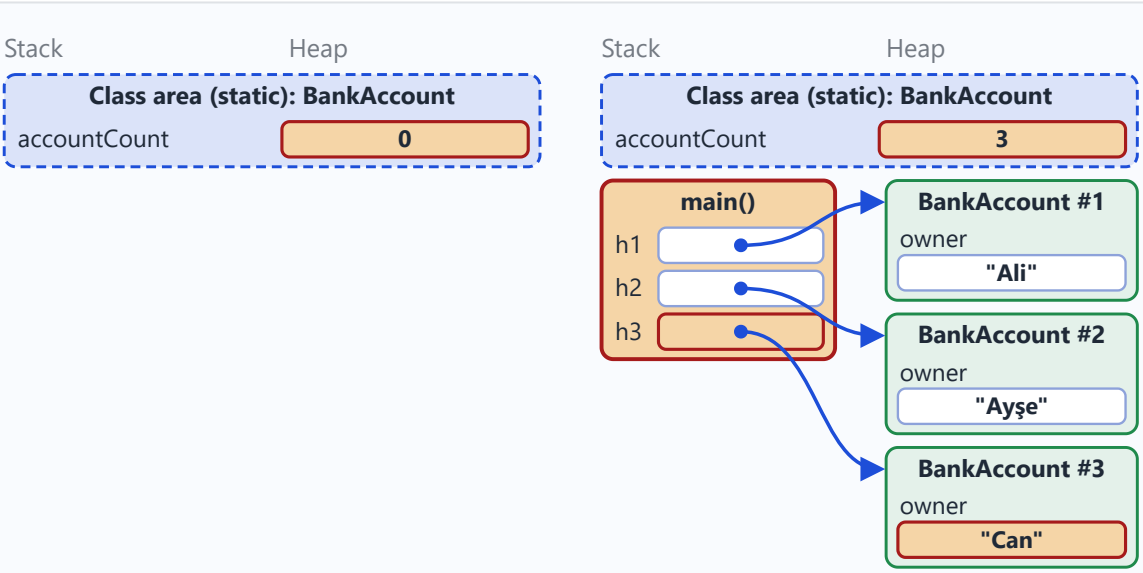


Figure 4.1.7: Left: the class has been loaded, the counter is 0, there is no object yet. Right: three objects have been created; each has its own `owner` field, but the counter is a single box and its value is 3.

The fourth and fifth steps do the same thing twice more. A separate `owner` field is created for each object, but the counter being incremented is always the same box: 2, then 3. The right part of Figure 4.1.7 draws this moment. On the heap there are three objects and three separate `owner` boxes; `accountCount`, on the other hand, is a single box outside the objects, in the region that belongs to the class.

In the sixth step `h2.owner` is read: an instance field is read through the object, and `Ayşe` is printed. In the seventh step the `static` method `getAccountCount()` is called with the class name. The frame of this method has no `this`, because the method was not called through any object. The method only

reads the `static` field and returns 3. In the eighth step the returned value is assigned to `count` and printed.

Because the `main` method is inside the `BankAccount` class, the line `h2.owner` compiles even though the field is `private`. `private` means "only code inside this class"; access is decided by the class, not by the object. Had the same line been written in the `main` of another class, a `has private access` error would have been the result.

4.1.4.3 Field declarations and final

The general form of a field declaration is: `[access] [static] [final] type name [= initialValue];`. The parts in square brackets are optional. The two classes below show different combinations.

```
public class Book {
  // Instance fields
  private String title;
  private String author;
  private int pageCount;
  // Class field
  private static int bookCount;
  // a final field (constant)
  public static final double PI = 3.14159265359;
}
```

```
public class Fields {
  public int a;                        // public
  int b = 1, c = 2;                    // default (package)
  private double x;                    // private
  private static int xx;               // static, private
  public static int y;                 // static, public
  public final int CONST1 = 5;         // cannot change
  private static final int CONST2 = 6; // static constant
}
```

The `Book` class has three instance fields, one class field and one constant. The `Fields` class lists the combinations of access modifiers with `static` and `final`. The line `int b = 1, c = 2;` declares two fields of the same type in one line and gives both an initial value. Because no access modifier is written, these two fields have package access.

A field declared `final` is given a value only once; it cannot be changed afterwards. The value is given either on the declaration line or in the constructor. A line that tries to assign a value later does not compile:

```
public class FinalAssign {
  public final int CONST1 = 5;
  void change() {
    CONST1 = 6;
  }
}
```

```
FinalAssign.java:4: error: cannot assign a value to final variable CONST1
    CONST1 = 6;
    ^
1 error
```

Class constants are usually declared `static final` and are written in capital letters separated by underscores: `MAX_VALUE`, `PI`. Because it is `static`, the constant has a single copy; because it is `final`, its value does not change. The `PI` in the `Book` class is defined only as an example. Java already has a ready-made constant that is more precise: `Math.PI`.

`CONST1` in the `Fields` class is `final` but not `static`. So every object has a separate copy of it, and the value of all of them is 5. Storing a constant that is the same for everyone separately in every object wastes memory; such constants are made `static final`.

4.1.4.4 Initializer blocks and initialization order

There are three ways to give a field an initial value: on the declaration line, in the constructor and in an initializer block. The third is used for preparation work that does not fit on one line.

Definition 4.1.10

An **initializer block** is a block of code written directly in the body of a class. A **static initializer** (`static { ... }`) runs only once, when the class is loaded into memory for the first time. An **instance initializer** (`{ ... }`) runs before the body of the constructor every time an object is created.

A static initializer is used to give a `static` field a complex initial value, for example to fill a table with a loop. An instance initializer gathers the work common to all constructors in one place. The order of the steps when an object is created is as follows:

1. If the class is being used for the first time, the initial values of the `static` fields and the `static` blocks run once, in the order in which they are written.
2. `new` allocates memory for the object; all fields receive default values.
3. The initial values on the field declaration lines and the instance initializer blocks run in the order in which they are written.

4. The body of the constructor runs.



Figure 4.1.8: The order of object creation. The first box runs only when the class is used for the first time; the other three run at every `new`.

The "order in which they are written" in the third step is important. Initial values on declaration lines and `{ }` blocks do not run as separate groups; they run in whatever order they are written in the class, from top to bottom. If an initializer block is written before a field declaration, the block runs first. The `Dog` class below has both kinds of initializer.

```
public class Dog {
  String name;                 // field
  Dog(String name) {          // constructor
    this.name = name;
    System.out.println(name);
  }
  String bark() {             // instance method
    return name + " woof woof!";
  }
  static {                     // static initializer
    System.out.println("Dogs!");
  }
  {                            // instance initializer
    System.out.println("Creating a dog.");
  }
  public static void main(String... strings) {
    var d1 = new Dog("Kara");
    var d2 = new Dog("Gece");
    d2.bark();
  }
}
```

```
Dogs!
Creating a dog.
Kara
Creating a dog.
Gece
```

Two notations in `main` may be new. `var` (since Java 10) infers the type of the local variable from the expression on the right: the line `var d1 = new Dog("Kara");` is the same as `Dog d1 = ...`. `String... strings` is the variable-arity parameter notation that stands in for `String[] strings`; for `main` the two are equivalent.

Step	Line	Call stack	Variables	New output
1	11: `System.out.println("Dogs!");`	static initializer	—	`Dogs!`
2	17: `var d1 = new Dog("Kara");`	main()	d1=?, d2=?	—
3	14: `System.out.println("Creating a dog.");`	main() → Dog("Kara")	d1=?, d2=?	`Creating a dog.`
4	4: `this.name = name;`	main() → Dog("Kara")	name="Kara"	—
5	5: `System.out.println(name);`	main() → Dog("Kara")	name="Kara"	`Kara`
6	18: `var d2 = new Dog("Gece");`	main()	d1=→#1, d2=→#2	`Creating a dog.` ⏎ `Gece`
7	19: `d2.bark();` 8: `return name + " woof woof!";`	main() → bark()	d1=→#1, d2=→#2	—

Table 4.1.7: The order in which initializers and constructors run in the `Dog` class.

In the first step, before `main` even starts, the `Dog` class is loaded, because `main` is a method of this class. While the class is being loaded, the `static` block runs and `Dogs!` is printed. In the second step `new Dog("Kara")` allocates space for the object; the `name` field is `null`. In the third step the instance initializer runs before the body of the constructor: `Creating a dog.` In the fourth and fifth steps the body of the constructor fills in the field and prints the name: `Kara`.

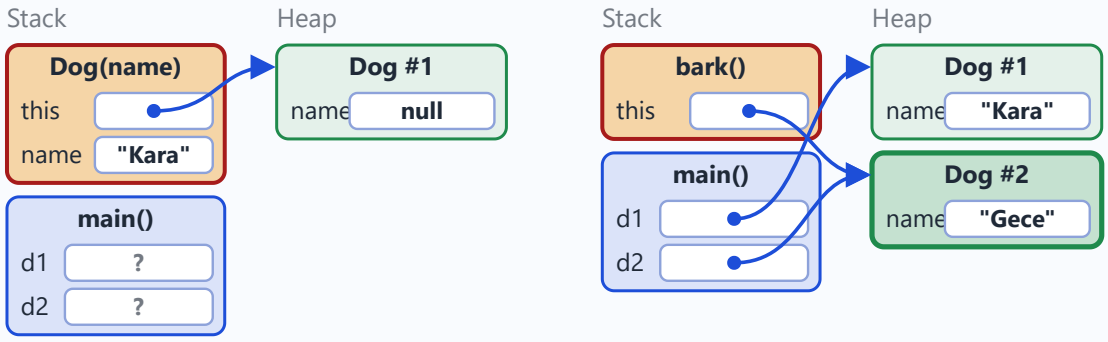


Figure 4.1.9: Left: the instance initializer has run and the constructor's frame is open; the object's `name` field is still `null`. Right: both objects are ready; `bark()` is running on object number 2.

The sixth step runs the initializer and the constructor again for the second object: `Creating a dog.` and `Gece`. The `static` block does not run this time, because the class is already loaded. In the seventh step `d2.bark()` is called. The method returns the text `"Gece woof woof!"`, but `main` neither assigns this value to a variable nor prints it. The returned value is lost unused and nothing appears on the screen. To make this text appear on the screen one would have to write `System.out.println(d2.bark());`.

Example 4.1.1

What does the following program print?

```
class A {
  static { System.out.print("S"); }
  { System.out.print("I"); }
  A() { System.out.print("C"); }
}
public class Problem3 {
  public static void main(String[] args) {
    System.out.print("M");
    new A();
    new A();
    System.out.println();
  }
}
```

`main` first prints `M`. Because class `A` is not the class of `main`, it is not loaded when the program starts; it is loaded at the first `new A()` expression. At that moment the `static` block runs once: `S`. Then the instance initializer (`I`) and the constructor (`C`) run for the first object. At the second `new A()` the class is already loaded; only `I` and `C` are printed. Result:

```
MSICIC
```

4.1.5 Methods

Methods define the behavior of a class. Defining a method, parameters, return values and the call stack are known from basic programming. The real question here is how methods relate to objects: whose data does a method work with, and how does it know?

4.1.5.1 Instance methods and class methods

As with fields, there are two kinds of methods. An **instance method** is tied to an object and uses that object's fields: `phone.call(...)`. A **class method** (`static` method) is tied to the class itself and is called through the class name: `Phone.updateOS(...)`.

```
public class Phone {
  private String model;
  private static String osVersion = "Version 1";

  // Instance method
  public void call(String number) {
    System.out.println(model + " calling: " + number);
  }

  // Instance method
  public void sendMessage(String to, String msg) {
    System.out.println(to + " <- " + msg);
  }

  // Class method (static)
  public static void updateOS(String newVersion) {
    osVersion = newVersion;
  }

  public static void main(String[] args) {
    Phone phone = new Phone();
    phone.model = "A54";
    phone.call("5551234567");
    phone.sendMessage("Ayşe", "I'm coming");
    Phone.updateOS("Version 2");
    System.out.println(osVersion);
  }
}
```

```
A54 calling: 5551234567
Ayşe <- I'm coming
Version 2
```

`call` and `sendMessage` are instance methods. The `model` in the body of `call` is the model of the phone that called the method. `updateOS`, on the other hand, is `static`: it changes the operating system version common to all phones and is not tied to any particular phone. For this reason a `static` method cannot directly access data specific to an object, such as `model`. Whose phone's model? The question has no answer, because the method was not called through any phone. The last line of `main` reads `osVersion` without writing the class name; because `main` is also a `static` method of the same class, this notation is valid.

4.1.5.2 A method that returns a value: Email

The header of a method summarizes what it does. The name of the `send` method below is `send`. It has three `String` parameters: the recipient, the subject and the body. Its return type is `boolean` and it reports whether the sending succeeded.

```
public class Email {
  public boolean send(String to, String subject,
            String body) {
    // a simulation, not a real send
    if (to == null || !to.contains("@")) {
      return false;               // failed
    }
    System.out.println(to + " <- " + subject);
    return true;                    // succeeded
  }

  public static void main(String[] args) {
    Email e = new Email();
    boolean s1 = e.send("ali@example.com", "Homework", "Hi");
    boolean s2 = e.send("ali.example.com", "Homework", "Hi");
    System.out.println(s1 + " " + s2);
  }
}
```

```
ali@example.com <- Homework
true false
```

The method does not send a real e-mail; it represents sending with a simulation. If the recipient address is `null` or does not contain `@`, it returns `false`. The `||` in the condition is the short-circuit operator: if `to == null` is true, the right side is never evaluated. Had the order been reversed, the call `to.contains("@")` would have thrown a `NullPointerException` for a `null` address. If the address is valid, the method prints the sending and returns `true`. `main` tries two sendings; in the second address there is a dot instead of `@`, so the second call returns `false` without printing anything to the screen.

Every path of a method that returns `boolean` must end with a `return`. Even a method whose body has not been written yet and contains only a comment does not compile:

```
public class MissingReturn {
  public boolean send(String to, String subject, String body) {
    // to be written later
  }
}
```

```
MissingReturn.java:4: error: missing return statement
  }
  ^
1 error
```

4.1.5.3 One piece of code, many objects

Instance fields and methods are declared in the class without the word `static`. Every object carries its own copy of the fields; for this reason objects can work with different data. To use an instance member an object is needed first: first `new`, then `object.method()`. The code of the method, however, is single for all objects. On every call, `this` decides whose data the method works with.

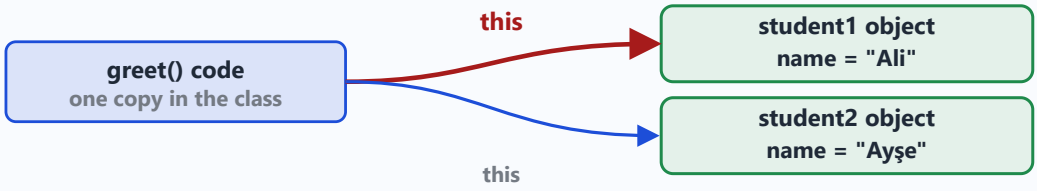


Figure 4.1.10: The code of `greet()` exists as a single copy in the class. In the call `student1.greet()`, `this` points to the first object; in the call `student2.greet()`, to the second.

	Instance	`static`
Belongs to	The object	The class
Call	`object.method()`	`ClassName.method()`
Object needed?	Yes	No
Has `this`?	Yes	No
Example	`str.length()`	`Math.sqrt(x)`

Table 4.1.8: Instance methods compared with `static` methods.

The last row of the table compares two familiar calls. `str.length()` gives the length of a text; `str` decides which text's length it is. `Math.sqrt(x)` gives the square root of a number. This job is not tied to any `Math` object; all the information needed is in the parameter `x`. Whether a method should be `static` is decided by this yardstick: if the method uses the fields of an object, it is an instance method; if it works only with its parameters, it can be `static`.

4.1.5.4 The dot operator and object parameters

Once an object has been created, its fields and methods are reached with the dot operator: `object.method(...)`, `object.area`. A method can also take another object as a parameter. In that case there are two objects inside the method: the object the method was called on and the object that came in as a parameter.

```
public class Phone {
  String number;
  public Phone(String number) {
    this.number = number;
  }
  public void call(Phone phone) {
    System.out.println(number + " calling: " + phone.number);
  }
  public void sendMessage(Phone phone, String msg) {
    System.out.println(phone.number + " <- " + msg);
  }
  public void openCamera() {
    System.out.println(number + " opened the camera.");
  }
  public static void main(String[] args) {
    Phone phone1 = new Phone("5551234567");
    Phone phone2 = new Phone("5051234567");
    phone1.call(phone2);
    phone2.sendMessage(phone1, "Hello!");
    phone1.openCamera();
  }
}
```

```
5551234567 calling: 5051234567
5551234567 <- Hello!
5551234567 opened the camera.
```

In the call `phone1.call(phone2)`, the bare `number` in the body of the method is the number of the object the method was called on, that is, `phone1`. `phone.number` is the number of the object that came in as a parameter, that is, `phone2`. That is why the first line has the form "5551234567 calling: 5051234567".

The second call must be read carefully. In the call `phone2.sendMessage(phone1, "Hello!")` the object calling the method is `phone2`, but the body uses only the number of the phone in the parameter: `phone.number`, that is, the number of `phone1`. This is why 5551234567 appears in the output. Although the method is named "send message", the number printed is the recipient's; the `number` that would show the sender is never used in the body. A bare field name in the body of a method and a field name reached through a parameter belong to different objects; if this distinction is overlooked, the wrong object's data is read.

The `private` fields of an object that comes in as a parameter can also be accessed, as long as the accessing code is inside the same class. `number` here has package access, but even if it were `private`, the notation `phone.number` would still compile. The access rule is per class, not per object.

4.1.5.5 The Rectangle class

The rectangle example from the comparison of the procedural and object-oriented approaches can now be written as a complete class.

```
public class Rectangle {
  private double length;     // rectangle's length
  private double width;    // rectangle's width

  // Constructor
  public Rectangle(double length, double width) {
    this.length = length;
    this.width = width;
  }

  public double area() {      // computes the area
    return length * width;
  }

  public double perimeter() {     // computes the perimeter
    return 2 * (length + width);
  }
}
```

The fields are `private`; they are used from outside only through the methods. The constructor takes the two sides and records them in the object. `area()` and `perimeter()` take no parameters, because the data they need is inside the object. The procedural counterpart would be `area(5.0, 3.0)`; with the object it is written `rect.area()`. The test class below creates two rectangles.

```
public class RectangleTest {
  public static void main(String[] args) {
    Rectangle rect = new Rectangle(5.0, 3.0);
    Rectangle square = new Rectangle(5.0, 5.0);
    double area = rect.area();
    double perimeter = rect.perimeter();
    System.out.println("Area: " + area
        + " Perimeter: " + perimeter);
    area = square.area();
    perimeter = square.perimeter();
    System.out.println("Area: " + area
        + " Perimeter: " + perimeter);
  }
}
```

```
Area: 15.0 Perimeter: 16.0
Area: 25.0 Perimeter: 20.0
```

Step	Line	Call stack	Variables	New output
1	3: `Rectangle rect = new Rectangle(5.0, 3.0);` 4: `Rectangle square = new Rectangle(5.0, 5.0);`	main()	rect=→#1, square=→#2	—
2	5: `double area = rect.area();`	main() → area()	rect=→#1, square=→#2	—
3	6: `double perimeter = rect.perimeter();`	main() → perimeter()	area=15.0	—
4	7: `System.out.println("Area: " + area` 8: `+ " Perimeter: " + perimeter);`	main()	area=15.0, perimeter=16.0	`Area: 15.0 Perimeter: 16.0`
5	9: `area = square.area();` 10: `perimeter = square.perimeter();`	main() → area()	area=25.0, perimeter=20.0	—
6	11: `System.out.println("Area: " + area` 12: `+ " Perimeter: " + perimeter);`	main()	area=25.0, perimeter=20.0	`Area: 25.0 Perimeter: 20.0`

Table 4.1.9: Trace of `RectangleTest`.

In the first step two objects are created; each has its own `length` and `width` values. In the second step `rect.area()` is called. In the method's frame `this` points to object number 1 and the method returns 5.0 × 3.0 = 15.0. In the third step `perimeter()` finds 2 × (5.0 + 3.0) = 16.0 for the same object. In the fourth step the two `double` values are appended to the text and printed; when `double` values are concatenated with `+`, they are written with their decimal part, in the form `15.0`.

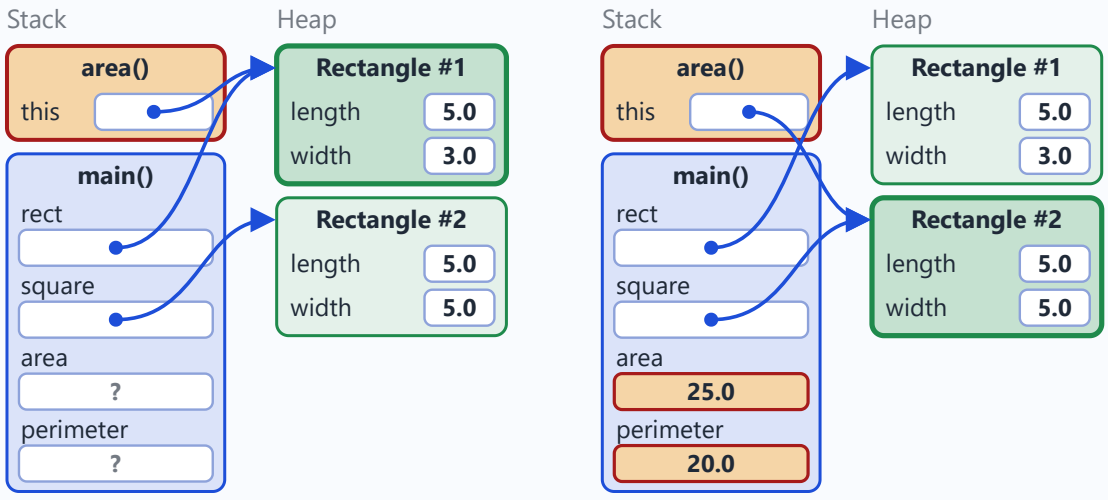


Figure 4.1.11: Left: `rect.area()` is running; `this` points to object number 1. Right: the same method called through `square`; `this` points to object number 2, and `area` and `perimeter` have received their new values.

In the fifth step the same two methods are called through `square`. This time `this` points to object number 2: 5.0 × 5.0 = 25.0 and 2 × (5.0 + 5.0) = 20.0. These values overwrite the old values of the variables `area` and `perimeter`. In the sixth step the second line is printed. That the `println` call is split over two lines is no problem: in Java a statement goes on until the semicolon.

The cost of both methods is constant: they do a few arithmetic operations and use no extra memory. Adding a field to the object, for example to keep the rectangle's color, does not affect these methods. In the procedural version, carrying the color information would require adding a new parameter to every call.

4.1.6 static members

The word `static` has appeared with both fields and methods. Gathering its rules in one place also explains the cause of the most common compile error.

4.1.6.1 Access without an object

Definition 4.1.11

A `static` **member** is a field or method declared with the `static` keyword that belongs not to any object but to the class itself. To reach a `static` member no object needs to be created; the class name and a dot are used: `ClassName.fieldName`, `ClassName.methodName(arguments)`.

```
public class MathUtil {
  // Static add method
  public static int add(int a, int b) {
    return a + b;
  }

  public static void main(String[] args) {
    int result = MathUtil.add(5, 3);
    System.out.println("Sum: " + result);
    System.out.println(Math.max(4, 9));
    System.out.println(Math.sqrt(16.0));
  }
}
```

```
Sum: 8
9
4.0
```

The call `MathUtil.add(5, 3)` is made without creating any `MathUtil` object. Because `main` is inside the same class, `add(5, 3)` could also have been written without the class name. `Math.max` and `Math.sqrt` are calls of the same kind: `static` methods of the `Math` class. Java's ready-made library has many examples of this: `Math.PI`, `Integer.parseInt("12")`, `Integer.MAX_VALUE`. The `main` with which every program starts is also `static`. When the program starts, the JVM (Java Virtual Machine) has not yet created any object; for it to call `main` without an object, the method has to belong to the class.

A `static` member can also be reached through an object: `h1.accountCount`. Java accepts this notation, but it is discouraged. A reader of the code assumes the member belongs to the object and thinks the counter of `h1` might differ from that of `h2`. The compiler treats this notation as `BankAccount.accountCount`; which object is used is irrelevant. If compiled with the `-Xlint` option, it also gives the warning `static variable should be qualified by type name`.

4.1.6.2 The rules of the static context

An instance method can use both instance members and `static` members directly. Such a method is always called through an object; that object is `this`, and the instance fields are its fields. A `static` class field is single anyway. A `static` method, on the other hand, has no `this`. This method can reach instance members only through an explicitly given object. Table 4.1.10 summarizes the four cases.

From	Instance field / method	`static` field / method
An instance method	✓ directly (`name`, `this.name`)	✓ directly
A `static` method (including `main`)	✗ only through an object (`s.name`)	✓ directly

Table 4.1.10: Which method can use which member directly?

Violating the rule is the compile error that beginners meet most often. Calling an instance method directly from `main`:

```
public class Greeter {
  private String name = "Ali";
  public void greet() {
    System.out.println("Hello " + name);
  }
  public static void main(String[] args) {
    greet();
  }
}
```

```
Greeter.java:7: error: non-static method greet() cannot be referenced from a static
context
    greet();
    ^
1 error
```

Reading an instance field directly from a `static` method:

```
public class Greeter2 {
  private String name = "Ali";
  public static void print() {
    System.out.println(name);
  }
}
```

```
Greeter2.java:4: error: non-static variable name cannot be referenced from a static
context
    System.out.println(name);
                       ^
1 error
```

The same phrase appears in both messages: `cannot be referenced from a static context`. A "`static` context" means a place where there is no `this`. The call `greet()` is really `this.greet()`, and the read of `name` is `this.name`. Because `main` has no `this`, the compiler cannot know whose method or field is meant. Writing `this` explicitly changes nothing either; the message then becomes `non-static variable this cannot be referenced from a static context`.

There are two fixes, and which one to choose depends on the design. If the method really needs the data of an object, the call is made through an object: `new Greeter().greet();`. If the method uses no object's data, it is made `static`. The second way is right only when the method is truly independent of any object. Making all methods and fields `static` just to silence the compile error turns the program back into the procedural style. Where each object should have its own data, a single shared copy remains.

4.1.7 References and this

The fact that a variable holds a reference, not an object, looks like a mere detail at first sight. But this detail determines how assignment, comparison and passing objects to methods behave.

4.1.7.1 Reference assignment and aliasing

In Java objects are handled through references. Assigning one reference variable to another does not copy the object; it copies only the reference. After the assignment two variables point to the same object.

> **Definition 4.1.12**
>
> An **alias** is any one of several reference variables that point to the same object. There is one object; a change made through one of the aliases is visible through the others too.

There are similar situations in everyday life. The label on a library shelf is not the book itself; it shows where the book is. A car may have two keys; both keys open the same car. If someone gets in with one key and pushes the seat back, the person who gets in with the other key finds the seat pushed back. The program below shows this situation with a `Car` class whose `brand` field is `private` and which has a getter and a setter.

```
public class Car {
  private String brand;

  public Car(String brand) {
    this.brand = brand;
  }

  public String getBrand() {
    return brand;
  }

  public void setBrand(String brand) {
    this.brand = brand;
  }

}
```

```
public static void main(String[] args) {
  Car car1 = new Car("Toyota");
  Car car2 = car1;
  System.out.println("Car 1: " + car1.getBrand());
  System.out.println("Car 2: " + car2.getBrand());
  car2.setBrand("Ford");
  System.out.println("Car 1: " + car1.getBrand());
  car1 = null;
  System.out.println("Car 2: " + car2.getBrand());
}
```

```
Car 1: Toyota
Car 2: Toyota
Car 1: Ford
Car 2: Ford
```

Step	Line	Call stack	Variables	New output
1	2: `Car car1 = new Car("Toyota");`	main()	car1=→#1, car2=?	—
2	3: `Car car2 = car1;`	main()	car1=→#1, car2=→#1	—
3	4: `System.out.println("Car 1: " + car1.getBrand());` 5: `System.out.println("Car 2: " + car2.getBrand());`	main()	car1=→#1, car2=→#1	`Car 1: Toyota ⏎ Car 2: Toyota`
4	6: `car2.setBrand("Ford");`	main() → setBrand("Ford")	car1=→#1, car2=→#1	—
5	7: `System.out.println("Car 1: " + car1.getBrand());`	main()	car1=→#1, car2=→#1	`Car 1: Ford`
6	8: `car1 = null;`	main()	car1=null, car2=→#1	—
7	9: `System.out.println("Car 2: " + car2.getBrand());`	main()	car1=null, car2=→#1	`Car 2: Ford`

Table 4.1.11: Trace of reference assignment. The line numbers are the lines of `main`.

In the first step a single object is created on the heap and `car1` points to it. The line `Car car2 = car1;` in the second step creates no new object. The reference in the box of `car1` is copied into the box of `car2`. There is still a single object on the heap; now two arrows lead to it. In the third step the brand is read through both variables and both give `Toyota`: the field being read is the same field.

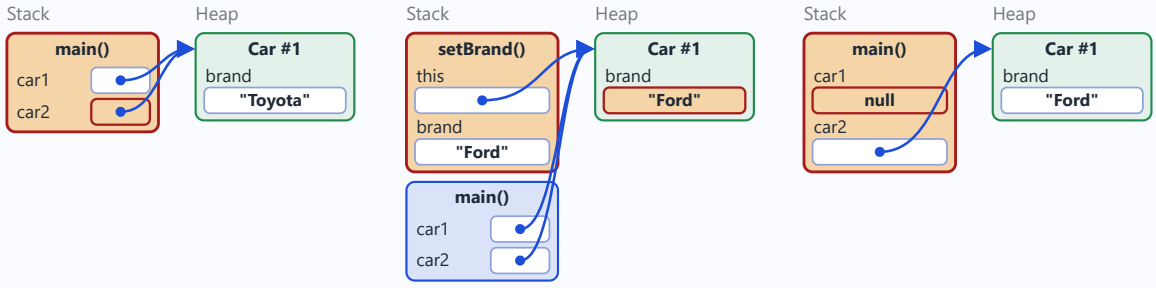


Figure 4.1.12: From left to right: after `car2 = car1` two arrows go to a single object; `car2.setBrand("Ford")` changes the shared object; `car1 = null` cuts only one arrow, and the object lives on through `car2`.

In the fourth step `car2.setBrand("Ford")` is called. The method's `this` is the object `car2` points to, that is, the single object. In the fifth step the brand read through `car1` is also `Ford`: the change is shared. This behavior is surprising if one does not know that the two variables point to the same object. The program thinks it has changed `car2`, but it has in fact also changed the object `car1` points to.

In the sixth step `car1 = null;` runs. This line does not delete the object; it only cuts the link between the variable `car1` and the object. The object stays where it is and `car2` keeps pointing to it. In the seventh step the brand read through `car2` is `Ford`.

For this reason the cost of assignment is also constant. However many fields the object has, `b = a` copies only one reference. If the object really has to be copied, a new object is created and the fields are transferred one by one; Java does not do this automatically with assignment.

The same rule applies when passing objects to methods. Java passes all arguments by value. If the argument is an object reference, the value copied is the reference. The method can change the shared object through this copy, but assigning a new object to its parameter does not affect the caller's variable:

```
class Box { int value; }
public class Problem4 {
  static void change(Box k) {
    k.value = 100;
    k = new Box();
    k.value = 7;
  }
  public static void main(String[] args) {
    Box x = new Box();
    x.value = 1;
    change(x);
    System.out.println(x.value);
  }
}
```

```
100
```

In the call `change(x)`, `k` receives a copy of the reference in `x`. `k.value = 100;` writes to the shared object; because `x` points to the same object, the change is visible from `main`. `k = new Box();`, however, points only the local copy to a new object. The `k.value = 7;` after that writes to the new object; the object of `x` is not affected. The program prints `100`.

4.1.7.2 null and NullPointerException

`null` means that a reference variable points to no object. Reading a field or calling a method through a `null` reference causes an error at run time, because there is no object to read from.

```
public class Empty {
  String name;

  public static void main(String[] args) {
    Empty b = null;
    System.out.println(b.name.length());
  }
}
```

```
Exception in thread "main" java.lang.NullPointerException: Cannot read field "name"
because "<local1>" is null
        at Empty.main(Empty.java:6)
```

The program compiles, because `b` has been given a value: `null`. When it runs, on line 6 it tries to read `b.name` and a `NullPointerException` is thrown. Since JDK 15 the message by default states the location of the error in detail: the `name` field could not be read because the variable is `null`. The notation `"<local1>"` in the message denotes local variable number 1 of `main`, that is, `b`. Had the program been compiled with debugging information (`javac -g`), the name of the variable would have appeared: `because "b" is null`.

If a local variable is used without ever being given a value, the situation is different: the program does not even compile (`variable sb might not have been initialized`). A `NullPointerException`, on the other hand, is an error the compiler cannot catch; it appears at run time. A field is its most frequent source: if a field of type `String` has not been given a value, its default value is `null`, and the first method called on it stops the program. The precaution is to check the reference before using it:

```
public class Empty2 {
  String name;

  public static void main(String[] args) {
    Empty2 b = new Empty2();
    if (b.name != null) {
      System.out.println(b.name.length());
    } else {
      System.out.println("name not set yet");
    }
    System.out.println(b.name.length());
  }
}
```

```
name not set yet
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()"
because "<local1>.name" is null
        at Empty2.main(Empty2.java:11)
```

The `if` block sees that `name` is `null` and takes the safe path. The last line, however, uses the same field without a check. This time the message says `"<local1>.name"`: it is not `b` but the `name` field of `b` that is `null`.



Figure 4.1.13: A `null` reference points to no object; when `b.name` is read, a `NullPointerException` is thrown. On the right, the garbage collector removes the object that no reference points to.

An object that no reference points to can never be reached from the program again. The memory of such objects is reclaimed by the JVM's **garbage collector** when the time comes. In Java there is no need to delete an object by hand, and no way to do so. Had `car2` also been set to `null` in the reference assignment trace, the last reference to the object would have been lost and the object would have become eligible for garbage collection. When the garbage collector will run is not known in advance; a program can only be sure that the object is no longer reachable.

4.1.7.3 The this keyword

Definition 4.1.13

`this` is, inside an instance method or constructor, the reference to the object the method was called on (in a constructor, the object being created). There is no `this` in `static` methods.

`this` is most often used in constructors, when the name of the field and the name of the parameter are the same: `this.name = name;`. In this case the bare name `name` refers to the parameter, because the nearest declaration is the parameter. The parameter **shadows** the field of the same name. To reach the field one writes `this.name`. What happens if `this` is forgotten?

```
public class Shadow {
  private String name;

  public Shadow(String name) {
    name = name;            // forgot this!
  }

  public static void main(String[] args) {
    Shadow g = new Shadow("Ali");
    System.out.println(g.name);
  }
}
```

```
null
```

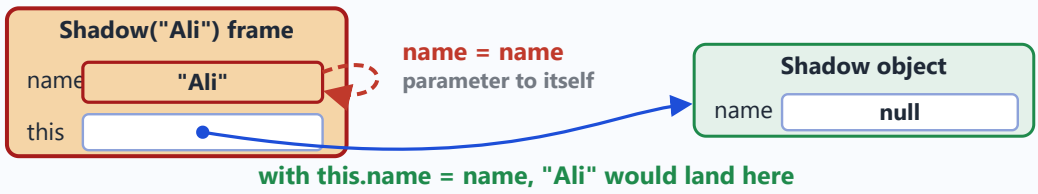


Figure 4.1.14: In the constructor, `name = name;` assigns the parameter to itself; the object's `name` field stays `null`. Had `this.name = name;` been written, `"Ali"` would have gone to the object's field.

Both `name`s on either side of the line `name = name;` are the parameter. The line assigns the parameter to itself and changes nothing. The object's `name` field is never touched; it stays at its default value `null`. The program prints `null`. The most dangerous part is this: the compiler gives no error for this line. The code compiles and runs; the error shows up only when the field is used, perhaps somewhere quite different, as a `NullPointerException`.

Had the field been `final`, the compiler would have caught the error. In the declaration `private final String name;` the field must receive a value in the constructor. Since `name = name;` gives the field no value, the compiler gives the message `variable name might not have been initialized`. Making fields that will never change `final` is therefore not only documentation but also a safety measure.

If there is no shadowing, writing `this` is optional. The bare `model` in a method body means `this.model` to the compiler. The trace below shows this step by step.

```
public class Student {
  private String name;

  public Student(String name) {
    this.name = name;
  }

  public void greet() {
    System.out.println("Hi, I am " + this.name);
  }

  public static void main(String[] args) {
    Student student1 = new Student("Ali");
    Student student2 = new Student("Ayşe");
    student1.greet();
    student2.greet();
  }
}
```

```
Hi, I am Ali
Hi, I am Ayşe
```

Step	Line	Call stack	Variables	New output
1	13: `Student student1 = new Student("Ali");` 4: `public Student(String name) {` 5: `this.name = name;`	main() → Student("Ali")	name="Ali"	—
2	14: `Student student2 = new Student("Ayşe");`	main()	student1=→#1, student2=→#2	—
3	15: `student1.greet();` 8: `public void greet() {` 9: `System.out.println("Hi, I am " + this.name);`	main() → greet()	student1=→#1, student2=→#2	`Hi, I am Ali`
4	16: `student2.greet();` 8: `public void greet() {` 9: `System.out.println("Hi, I am " + this.name);`	main() → greet()	student1=→#1, student2=→#2	`Hi, I am Ayşe`

Table 4.1.12: Which object `this` points to in each call.

In the first step the constructor runs; `this` is the newly created object, and `"Ali"` is written to the field with `this.name = name;`. In the second step the second object is created with the same constructor; at that moment `this` points to object number 2. In the third step `student1.greet()` is called. The object to the left of the dot becomes the method's `this`, and `Hi, I am Ali` is printed. In

the fourth step the same code runs with a different `this` and prints `Hi, I am Ayşe`. In this method, writing just `name` instead of `this.name` would give the same result, because the method has no parameter called `name`.

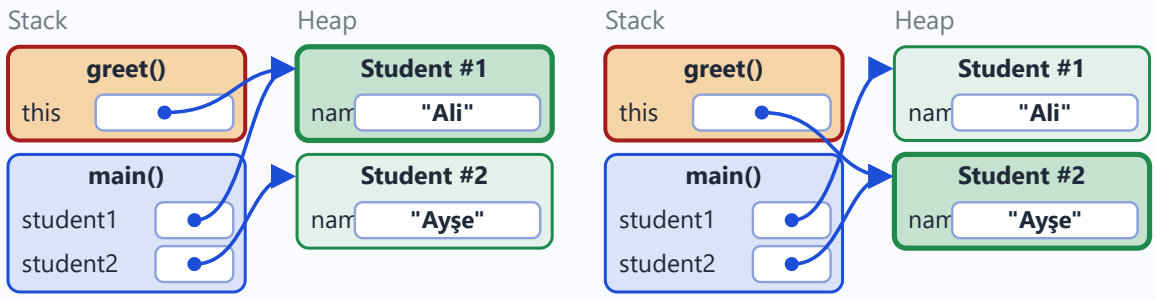


Figure 4.1.15: Left: `student1.greet()`; `this` points to object number 1. Right: `student2.greet()`; the same method, but `this` points to object number 2.

4.1.7.4 Passing the object with this

Because `this` is a reference, it can also be given to another method as an argument. This is the way to go when an object wants to introduce itself to another object.

```
public class Team {
  private String name;
  public Team(String name) {
    this.name = name;
  }
  public void addPlayer(Player player) {
    player.joinTeam(this);   // this object
  }
  public String getName() {
    return name;
  }
  public static void main(String[] args) {
    Team team = new Team("Giresunspor");
    Player player = new Player("Mert");
    team.addPlayer(player);
  }
}
```

In the same file, a second class that is not `public`:

```
class Player {
  private String name;
  Player(String name) {
    this.name = name;
  }
  void joinTeam(Team team) {
    String t = team.getName();
    System.out.println(name + " -> " + t);
  }
}
```

```
Mert -> Giresunspor
```

In the call `team.addPlayer(player)`, the method's `this` is the `team` object. The `player.joinTeam(this)` in the body of the method hands this reference to the player. The `team` parameter of `joinTeam` now points to the same team object and reads the name with `team.getName()`. At this moment there are two separate `this` values in two frames: the `this` of `addPlayer` is the team, and the `this` of `joinTeam` is the player. The bare `name` in `joinTeam` is also the player's name. The output is `Mert -> Giresunspor`.

The `Player` class is defined inside the file `Team.java`, without `public`. A file can contain only one `public` class, but helper classes may be in the same file. These classes have package access; the other classes in the same folder can use them.

4.1.7.5 Constructor chaining with this(...)

If a class has several constructors, most of them do the same things. Instead of copying the same assignments into every constructor, constructors can hand work over to each other.

Definition 4.1.14

`this(...)` is the notation that calls another constructor of the same class from inside a constructor. The sequence of constructors formed this way is called **constructor chaining**. `this(...)` must be the first statement of the constructor.

```
class Employee {
  private String name;
  private double salary;
  Employee(String name) {
    this(name, 20000.0);   // same-class constructor
  }
  Employee(String name, double salary) {
    this.name = name;        // init in one place
    this.salary = salary;
  }
}
```

```
Elif 20000.0
```

The common initial assignments are gathered in a single constructor; here this constructor is `Employee(String, double)`. The other constructor adds the missing argument and delegates to it: `this(name, 20000.0)`. Every employee created without a salary starts with 20000.0. If the default salary changes, a single number changes. If the assignment rules change, for example if one wants to check that the name is not empty, the check is written in a single constructor.

The chain has two rules, and the compiler checks both. `this(...)` must be on the first line of the constructor:

```
public class ThisFirst {
  int x;
  ThisFirst() { x = 1; this(2); }
  ThisFirst(int x) { this.x = x; }
}
```

```
ThisFirst.java:3: error: call to this must be first statement in constructor
  ThisFirst() { x = 1; this(2); }
                       ^
1 error
```

Constructors cannot call each other in a cycle; such a chain would never end:

```
public class ThisLoop {
  ThisLoop() { this(1); }
  ThisLoop(int x) { this(); }
}
```

```
ThisLoop.java:3: error: recursive constructor invocation
  ThisLoop(int x) { this(); }
                    ^
1 error
```

However many constructors the chain passes through, a single object is built. The body of the called constructor finishes first; then the remaining lines of the caller run. The trace below shows this.

```
class Book {
  String name;
  String author;
  int pageCount;
  public Book(String name) {
    this.name = name;
  }
  public Book(String name, String author) {
    this(name);  // constructor chaining
    this.author = author;
  }
}
// in main:
Book k2 = new Book("Nutuk", "Atatürk");
System.out.println(k2.name + " / "
    + k2.author + " / " + k2.pageCount);
```

```
Nutuk / Atatürk / 0
```

Step	Line	Call stack	Variables	New output
1	14: `Book k2 = new Book("Nutuk", "Atatürk");`	main()	k2=?	—
2	8: `public Book(String name, String author) {`	main() → Book(name, author)	name="Nutuk", author="Atatürk"	—
3	9: `this(name); // constructor chaining`	main() → Book(name, author) → Book(name)	name="Nutuk"	—
4	6: `this.name = name;`	main() → Book(name, author) → Book(name)	name="Nutuk"	—
5	10: `this.author = author;`	main() → Book(name, author)	name="Nutuk", author="Atatürk"	—
6	14: `Book k2 = new Book("Nutuk", "Atatürk");`	main()	k2=→Book	—

Step	Line	Call stack	Variables	New output
7	15: `System.out.println(k2.name + " / "` 16: `+ k2.author + " / " + k2.pageCount);`	main()	k2=→Book	`Nutuk / Atatürk / 0`

Table 4.1.13: Trace of the constructor chain in the call `new Book("Nutuk", "Atatürk")`.

In the first step `new` allocates a new `Book` object on the heap; the fields receive their default values: `null`, `null`, `0`. In the second step, because the arguments are two `String`s, the two-parameter constructor is chosen; `this` points to the new object. In the third step `this(name)` runs and the one-parameter constructor is called for the same object. The stack now holds two constructor frames one above the other; the `this` of both is the same object.

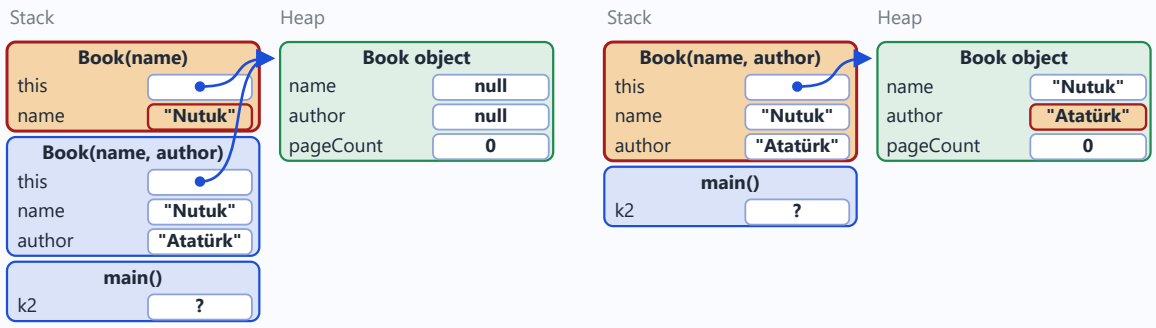


Figure 4.1.16: Left: the deepest moment of the chain; two constructor frames are open and the `this` of both points to the same object. Right: the inner constructor has finished and the outer constructor has filled in the `author` field.

In the fourth step the inner constructor writes `"Nutuk"` to the `name` field with `this.name = name;` and ends. In the fifth step the outer constructor resumes where it left off and runs its remaining line: `this.author = author;`. In the sixth step both constructors have finished; the reference that is the value of the `new` expression is assigned to `k2`. In the seventh step the fields are read. Because no constructor gave `pageCount` a value, its default value is printed: `Nutuk / Atatürk / 0`.

There is also a side effect here. The initial values on the field declaration lines and the instance initializers run once, in the constructor of the chain that does not start with `this(...)`, immediately before the body of that constructor. A constructor that starts with `this(...)` leaves this job to the constructor at the end of the chain. So however long the chain is, the initializers run exactly once for each object.

4.1.8 Comparing objects: == and equals

Comparing two `int` variables with `==` compares the numbers. Comparing two reference variables with `==`, however, compares not the objects but the references in the variables' boxes. The question is not "do these two objects have the same content?" but "do these two variables point to the same object?"

Definition 4.1.15

Reference equality means two references pointing to the same object and is tested with `==`. **Content equality** means two objects carrying the same value in meaning and is tested with the `equals()` method. What `equals` counts as "the same" is decided by the programmer who writes the class.

In the program below, the `Person` class writes its own `equals` method: it counts two people as equal if their names and ages are the same. The `Person0` class, on the other hand, does not write `equals`. The last part of the program reads a word from the keyboard and compares it with a text literal.

```
import java.util.Objects;
import java.util.Scanner;
class Person {
  private String name;
  private int age;
  Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Person)) return false;
    Person k = (Person) o;
    return age == k.age && Objects.equals(name, k.name);
  }
  @Override
  public int hashCode() {
    return Objects.hash(name, age);
  }
  @Override
  public String toString() {
    return name + " (" + age + ")";
  }
}
class Person0 {
  String name;
  Person0(String name) { this.name = name; }
}
public class Equality {
  public static void main(String[] args) {
    Person k1 = new Person("Ali", 20);
    Person k2 = new Person("Ali", 20);
    Person k3 = k1;
    System.out.println(k1 == k3);
    System.out.println(k1 == k2);
    System.out.println(k1.equals(k2));
    System.out.println(k1.hashCode() == k2.hashCode());
    System.out.println(k1);
    Person0 a = new Person0("Ali");
    Person0 b = new Person0("Ali");
    System.out.println(a.equals(b));
    Scanner in = new Scanner(System.in);
    String input = in.next();
    System.out.println(input == "Java");
    System.out.println(input.equals("Java"));
  }
}
```

When `Java` is typed at the keyboard, the program prints:

```
true
false
true
true
Ali (20)
false
false
true
```

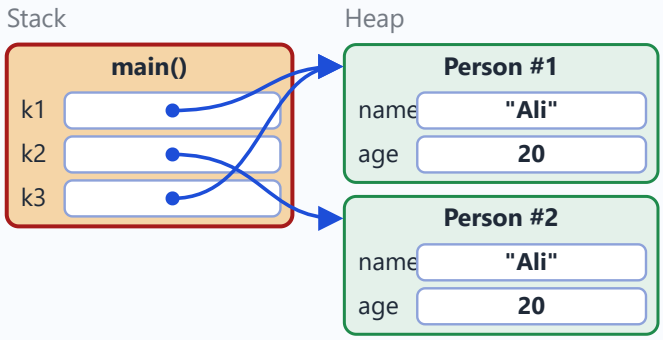


Figure 4.1.17: `k1` and `k3` point to the same `Person` object, `k2` to a separate object with the same content.

Expression	Result	Why
`k1 == k3`	`true`	The two variables point to the same object.
`k1 == k2`	`false`	Different objects; the references differ even though the contents are the same.
`k1.equals(k2)`	`true`	`equals` has been overridden; the name and age are the same.

Table 4.1.14: Comparison results for three references.

The methods `equals`, `hashCode`, `toString` and `getClass` come to every class from `Object`, the topmost class of Java. The `Object` version of `equals` compares only references; that is, it behaves like `==`. Because the `Person0` class does not write `equals`, `a.equals(b)` gives `false`: there are two separate objects and the `Object` version does not look at the content. If the objects of a class are to be compared by content, `equals` must be overridden in that class.

The `equals` in `Person` works in three steps. First it checks with `this == o` whether it is the same object; if so, it returns `true` immediately. Then it checks with `instanceof` whether the incoming object is a `Person`; if not, or if `o` is `null`, it returns `false`. In the last step it compares the fields. `age` is an `int`, so it is compared with `==`; `name` is a `String`, so it is compared with `Objects.equals`. This helper method also works safely when one of the two sides is `null`. The type of the parameter is `Object`, not `Person`. Had the type been written `Person`, the method would not have overridden the `equals` of `Object` but only overloaded it; the `@Override` annotation catches this error at compile time.

When `equals` is overridden, `hashCode` must be written too. The rule is: two objects that are equal according to `equals` must have equal `hashCode` values. The fourth line of the program verifies this. If the rule is broken, collections such as `HashMap` and `HashSet` look for equal objects in different places and fail to find them. `Objects.hash(name, age)` produces a value from the same fields used in `equals`. `toString` gives the text form of the object; `println(k1)` calls this method automatically and prints `Ali (20)`. Had `toString` not been written, a text made of the class name and a number, such as `Person@1b6d3586`, would have been seen.

`String` and the wrapper classes are also objects and obey the same rule. The last two lines of the program show this. The `"Java"` read from the keyboard is a new `String` object created at run time. The `"Java"` literal in the code is another object. Although their contents are the same, `==` gives `false` and `equals` gives `true`. That `==` sometimes gives `true` when two text literals are compared is because Java stores the same literal as a single object. This is not a rule but an implementation detail; texts are always compared with `equals`.

4.1.9 Wrapper classes

4.1.9.1 The object counterpart of a primitive type

> **Definition 4.1.16**
>
> A **wrapper class** is a class that carries the value of a primitive type inside an object. Each of the eight primitive types in Java has a wrapper class.

Primitive type	Wrapper class	Primitive type	Wrapper class
`byte`	`Byte`	`float`	`Float`
`short`	`Short`	`double`	`Double`
`int`	`Integer`	`char`	`Character`
`long`	`Long`	`boolean`	`Boolean`

Table 4.1.15: Primitive types and their wrapper classes.

Most of the names are the primitive type with a capital first letter; the two exceptions are `Integer` and `Character`. These classes are needed for three reasons:

- **Sometimes an object is required.** Generic collections such as `ArrayList<Integer>` cannot take a primitive type; the notation `ArrayList<int>` does not compile.
- **The "no value" case.** `null` cannot be assigned to a primitive type; it can be assigned to a variable of the wrapper type. If the grade of an exam has not been entered yet, `Integer grade = null;` shows this explicitly. With `int` and a 0, "scored zero" and "no grade yet" could not be told apart.

- **Helper methods and constants.** `Integer.parseInt("12")` converts a text to a number, `Integer.MAX_VALUE` gives the largest `int` value, and `Character.isDigit(c)` tells whether a character is a digit.

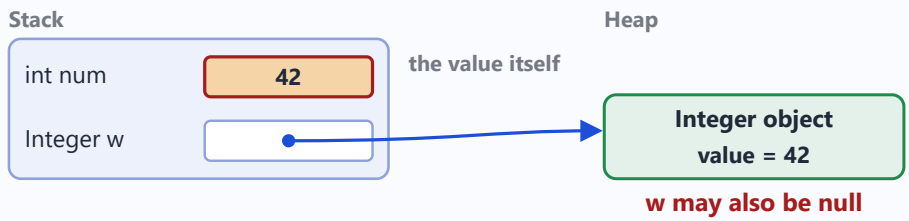


Figure 4.1.18: An `int` value is kept directly on the stack. An `Integer` variable points to an object on the heap; the value is inside that object, and `null` can also be assigned to the variable.

Wrappers have a price. As seen in Figure 4.1.18, an `int` value sits in the variable's box; an `Integer` requires a separate object, a reference leading to it and an indirect access when read. In a loop that processes millions of numbers this difference is measurable. For this reason the primitive type is preferred unless there is a need.

4.1.9.2 Autoboxing and unboxing

Definition 4.1.17

Autoboxing is the conversion of a primitive value into a wrapper object, when needed, through a call the compiler inserts. **Unboxing** is the conversion of the value in a wrapper object back into the primitive type.

```
public class Wrappers {
  public static void main(String[] args) {
    // an int variable
    int num = 42;
    // using the Integer wrapper
    Integer w1 = Integer.valueOf(num);
    // Autoboxing (automatic)
    Integer w2 = num;
    // Unboxing (explicit and automatic)
    int n2 = w2.intValue();
    int n3 = w2;
    System.out.println(n2 + n3);
    System.out.println(w1.equals(w2));
    System.out.println(Integer.parseInt("123") + 1);
    System.out.println(Integer.MAX_VALUE);
    System.out.println(Character.isDigit('7'));
    System.out.println(Double.parseDouble("2.5") * 2);
    Integer empty = null;      // an int can't be null
    System.out.println(empty);
  }
}
```

```
84
true
124
2147483647
true
5.0
null
```

The line `Integer w2 = num;` assigns an `int` to an `Integer` variable. The compiler silently turns this line into the call `Integer.valueOf(num)`; this is autoboxing. The line `int n3 = w2;` is turned into the call `w2.intValue()`; this is unboxing. The explicit forms (`Integer.valueOf`, `intValue()`) and the automatic forms produce the same code.

The first line of the output is `42 + 42 = 84`. The contents of the two wrappers are the same: `true`. The text `"123"` is converted to a number and 1 is added: `124`. The fourth line is the largest `int` value: `2147483647`, that is, $2^{31} - 1$. `'7'` is a digit: `true`. `"2.5"` is converted to `double` and multiplied by 2: `5.0`. On the last line an `Integer` with the value `null` is printed. `println` prints a `null` reference as the word `null`; since there is no unboxing here, there is no error either. Assigning the same value to an `int` does not compile:

```
public class IntNull {
  public static void main(String[] args) {
    int empty = null;
  }
}
```

```
IntNull.java:3: error: incompatible types: <null> cannot be converted to int
    int empty = null;
                ^
1 error
```

4.1.9.3 The Integer cache and the null trap

Automatic conversions shorten the code, but they hide two traps. Both come from forgetting that wrappers are objects.

```
import java.util.ArrayList;
public class Pitfalls {
  public static void main(String[] args) {
    ArrayList<Integer> list = new ArrayList<>();
    list.add(5);                // autoboxing: 5 -> Integer
    int first = list.get(0);      // unboxing: Integer -> int
    System.out.println(first + 1);
    Integer a = 127, b = 127;
    Integer c = 128, d = 128;
    System.out.println(a == b);
    System.out.println(c == d);
    System.out.println(c.equals(d));
    Integer empty = null;
    int x = empty;                 // NullPointerException
  }
}
```

```
6
true
false
true
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"java.lang.Integer.intValue()" because "<local7>" is null
        at Pitfalls.main(Pitfalls.java:14)
```

The first three lines are fine: `list.add(5)` boxes the 5, `list.get(0)` returns the object, and it is unboxed when assigned to `int first`; `first + 1` is 6. The next two comparisons, however, are surprising: `a == b` gives `true` and `c == d` gives `false`. Although the values are equal, why does

the second comparison come out false? The trace below shows it step by step.

```
Integer a = 127, b = 127;
Integer c = 128, d = 128;
System.out.println(a == b);
System.out.println(c == d);
System.out.println(c.equals(d));
int t = c + d;
System.out.println(t);
Integer empty = null;
int x = empty;
```

Step	Line	Call stack	Variables	New output
1	1: `Integer a = 127, b = 127;`	—	a=→cache(127), b=→cache(127)	—
2	2: `Integer c = 128, d = 128;`	—	a=→cache, b=→cache, c=→#1, d=→#2	—
3	3: `System.out.println(a == b);`	—	a=→cache, b=→cache	`true`
4	4: `System.out.println(c == d);`	—	c=→#1, d=→#2	`false`
5	5: `System.out.println(c.equals(d));`	—	c=→#1, d=→#2	`true`
6	6: `int t = c + d;`	—	c=→#1, d=→#2, t=256	—
7	7: `System.out.println(t);`	—	t=256	`256`
8	8: `Integer empty = null;`	—	t=256, empty=null	—
9	9: `int x = empty;`	—	empty=null, x=?	`Exception in thread "main" ⏎ java.lang.NullPointerException`

Table 4.1.16: Trace of the `Integer` cache and of unboxing `null`.

In the first step `Integer a = 127` turns into the call `Integer.valueOf(127)` through autoboxing. The `valueOf` method does not create a new object every time for values between −128 and 127; the objects for these values are ready in a cache. `a` and `b` receive the same object from the cache. In the second step 128 is outside this range; `valueOf` creates a new object on each call, and `c` and `d` point to two separate objects.

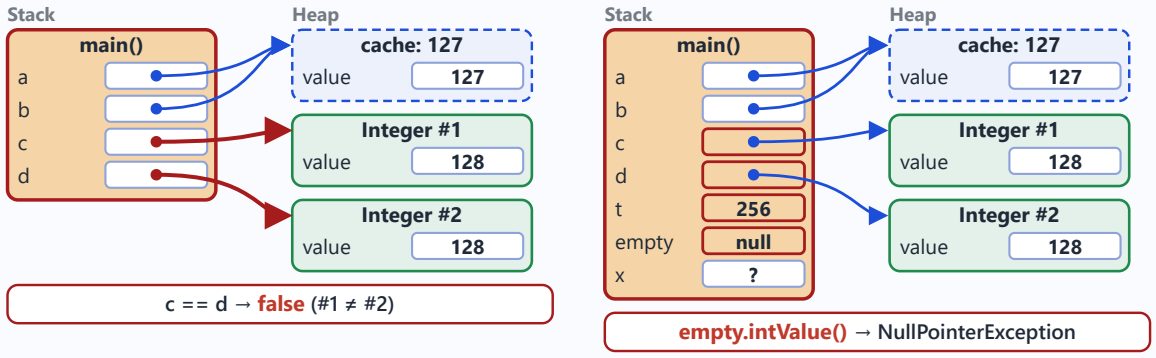


Figure 4.1.19: Left, `c == d` : the values are the same (128), but the two arrows go to two separate objects; `a` and `b` share the single object in the cache. Right, `empty` points to no object; when `int x = empty` tries to unbox, a `NullPointerException` is thrown.

In the third and fourth steps `==` compares references. The result is `true` because `a` and `b` point to the same object, and `false` because `c` and `d` point to different objects. The values are the same, the objects are different. In the fifth step `c.equals(d)` compares the `int` values inside the objects and gives `true`. The trap is here: a trial with small numbers gives the impression that `==` works correctly; when the code meets large numbers it silently produces a wrong result. Wrapper objects are always compared with `equals`.

In the sixth step `int t = c + d;` runs. The `+` operator does not work with objects; both objects are unboxed (`intValue()`) and 128 + 128 = 256 is found. `t` is a primitive `int` and its value sits in its box on the stack. The same rule applies in comparison: if an `Integer` is compared with an `int` using `==`, the `Integer` is unboxed and the numbers are compared. In the seventh step 256 is printed.

In the eighth step `Integer empty = null;` runs. `null` can be assigned to a wrapper variable. In the ninth step `int x = empty;` is quietly turned into the call `empty.intValue()`. There is no object to point to and a `NullPointerException` is thrown. The message names this hidden call explicitly: `Cannot invoke "java.lang.Integer.intValue()"`. That a line in which no method call is visible in the code gives this error is the most dangerous side of automatic unboxing.

Common mistake

Comparing wrapper objects with `==`. `Integer` values in the range −128…127 seem to give the right result with `==`; outside the range the same values give `false`. The upper limit of the cache can be raised with a JVM setting, so relying on the limit is not right either. Content is always compared with `equals`.

4.1.10 Records and enums

Some classes are written only to carry a few pieces of data together: the two coordinates of a point, the name and number of a student. Some types, on the other hand, take one of a known, limited set of values: the days of the week, the seasons. Java offers a short and safe notation for each of these cases.

4.1.10.1 Records

Definition 4.1.18

A **record** (Java 16 and later) is a kind of class designed to write classes that only carry data in a short form. All the fields of a record are `private final` and it has no setters; its objects are therefore **immutable**.

```
record Point(double x, double y) { }
```

This single line defines a record with two `double` components named `x` and `y`. From this definition the compiler generates two `private final` fields and a constructor with two parameters. It also adds the accessors `x()` and `y()`, an `equals` that compares by content, a matching `hashCode` and a readable `toString`. The names of the accessors do not start with `get`; they carry the name of the component. Writing the same class by hand would take about 30 lines.

```
public class RecordTest {
  public static void main(String[] args) {
    Point p1 = new Point(1.0, 2.5);
    Point p2 = new Point(1.0, 2.5);
    boolean same = p1.equals(p2);
    System.out.println(same);
    System.out.println(p1 == p2);
    System.out.println(p1);
    System.out.println(p1.x() + p1.y());
  }
}
```

```
true
false
Point[x=1.0, y=2.5]
3.5
```

The two points are created with the same coordinates. `equals` gives `true` because it compares content, and `==` gives `false` because it compares references. `println(p1)` calls the `toString` that the record generates: `Point[x=1.0, y=2.5]`. The last line adds the two values read with the accessors: 3.5.

If an attempt is made to change a field of a record, the program does not compile:

```
record Point(double x, double y) { }
public class RecordChange {
  public static void main(String[] args) {
    Point n = new Point(1.0, 2.5);
    n.x = 3.0;
  }
}
```

```
RecordChange.java:5: error: x has private access in Point
    n.x = 3.0;
     ^
1 error
```

If a record needs a new value, a new record is created. A record can also have its own methods. To add a check to the constructor, a compact constructor is defined without repeating the component list:

```
record Point(double x, double y) {
  Point {
    if (Double.isNaN(x) || Double.isNaN(y))
      throw new IllegalArgumentException("NaN not allowed");
  }
  double distance() { return Math.sqrt(x * x + y * y); }
  Point shift(double dx) { return new Point(x + dx, y); }
}
public class RecordExtra {
  public static void main(String[] args) {
    Point n = new Point(3.0, 4.0);
    Point m = n.shift(1.0);
    System.out.println(n + " " + n.distance());
    System.out.println(m);
    System.out.println(n.hashCode() == new Point(3.0, 4.0).hashCode());
  }
}
```

```
Point[x=3.0, y=4.0] 5.0
Point[x=4.0, y=4.0]
true
```

The `Point { ... }` block is the compact constructor. The compiler adds the parameters, and the block does only the check; the assignment to the fields is done automatically after the block ends. `distance()` computes the distance of the point from the origin: $\sqrt{(3^2 + 4^2)} = 5.0$. `shift` does not change the object; it returns a new `Point`, and `n` keeps its old value. The last line verifies that the `hashCode` values of two equal records are also equal.

4.1.10.2 enum

Definition 4.1.19

An **enum** (enumeration type) is a special kind of class that represents a fixed, limited set of values. Each value is the one and only object of that type for the whole program.

```
enum Weekday {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
  FRIDAY, SATURDAY, SUNDAY
}
public class DayTest {
  public static void main(String[] args) {
    Weekday[] allDays = Weekday.values();
    Weekday thu = Weekday.valueOf("THURSDAY");
    System.out.println(allDays.length + " days");
    System.out.println(thu + " " + thu.ordinal());
    for (Weekday day : allDays) {
      boolean weekend = day == Weekday.SATURDAY
          || day == Weekday.SUNDAY;
      if (weekend) {
        System.out.println(day + ": weekend");
      }
    }
  }
}
```

```
7 days
THURSDAY 3
SATURDAY: weekend
SUNDAY: weekend
```

The `Weekday` type has only seven values and no other value can be created. It would also have been possible to keep the days of the week as numbers 0–6 in an `int`, but then the compiler could not catch a meaningless assignment such as `day = 9`. Only values of an `enum` type can be assigned to a variable of that type.

Three useful methods come with every `enum` type. `values()` returns all the values as an array in declaration order; a loop can be run over it. `valueOf("THURSDAY")` converts a text into the value of the same name. `ordinal()` gives the value's position number starting from 0: `THURSDAY` is 3 because it is the fourth value. Because every value is a single object, `enum` values are compared safely with `==`; the weekend check in the loop is written this way.

Because an `enum` is a class, it can have fields, a constructor and methods. In the example below each planet carries its own gravity factor. The program also shows the use of `enum` with `switch` and that `valueOf` looks for the name exactly.

```
enum Planet {
  MERCURY(0.38), EARTH(1.0), MARS(0.38);
  private final double gravity;
  Planet(double gravity) { this.gravity = gravity; }
  double weight(double mass) { return mass * gravity; }
}
public class EnumFeatures {
  public static void main(String[] args) {
    Planet g = Planet.MARS;
    switch (g) {
      case EARTH -> System.out.println("home");
      case MARS -> System.out.println("red");
      default -> System.out.println("other");
    }
    System.out.println(g.weight(70.0));
    System.out.println(Planet.valueOf("Mars"));
  }
}
```

```
red
26.6
Exception in thread "main" java.lang.IllegalArgumentException: No enum constant
Planet.Mars
        at java.base/java.lang.Enum.valueOf(Enum.java:293)
        at Planet.valueOf(EnumFeatures.java:1)
        at EnumFeatures.main(EnumFeatures.java:16)
```

The parentheses next to the values are arguments passed to the constructor. The constructor of an `enum` is always `private`; `new Planet(...)` cannot be written from outside. In `switch`, the type name is not written in the `case` labels: `case MARS`. 70 × 0.38 = 26.6 is computed. On the last line `valueOf("Mars")` is called; since the name of the value is `MARS`, no match is found and an `IllegalArgumentException` is thrown. `valueOf` is case-sensitive.

4.1.11 Inner classes

Definition 4.1.20

An **inner class** is a class defined inside another class. An inner class can directly access the fields and methods of the outer class, including the `private` ones.

Inner classes are used to gather, inside a class, helper classes that serve only that class. The most common example is the listener classes in graphical interfaces, which carry the code that runs when a button is clicked. There are four kinds of inner class, depending on where they are defined.

Kind	Where defined?	How created?
`static` inner class	In the class body, with `static`	Without an outer object: `new Outer.StaticInner()`
Member inner class	In the class body, without `static`	Through an outer object: `outer.new MemberInner()`
Local inner class	Inside a method	Only from inside that method
Anonymous inner class	Inside an expression, nameless	A single object where it is defined; implements an interface or abstract class

Table 4.1.17: Kinds of inner classes.

4.1.11.1 static inner class

```
public class Outer {
  // a static inner class inside Outer
  static class StaticInner {
    void display() {
      System.out.println("Static Inner Class");
    }
  }

  public static void main(String[] args) {
    // access without an Outer instance
    Outer.StaticInner staticInner = new Outer.StaticInner();
    staticInner.display();
  }
}
```

```
Static Inner Class
```

A `static` inner class is like an ordinary class placed inside the outer class. It can be created without an object of the outer class: `new Outer.StaticInner()`. Its full name is `Outer.StaticInner`; the outer class acts here as a namespace. Because it is not tied to an outer object, it can directly access only the `static` members of the outer class. If it tries to read an instance field of the outer class, the familiar message results:

```
public class InnerStaticOuter {
  private int count = 5;
  static class StaticInner {
    void display() { System.out.println(count); }
  }
}
```

```
InnerStaticOuter.java:4: error: non-static variable count cannot be referenced from a
static context
    void display() { System.out.println(count); }
                                       ^
1 error
```

4.1.11.2 Member inner class

```
public class Outer {
  private int count = 5;          // outer's private field

  class MemberInner {            // member inner class
    void display() {
      System.out.println("Member Inner Class, count = " + count);
    }
  }

  public static void main(String[] args) {
    // access via an Outer instance
    Outer outer = new Outer();
    Outer.MemberInner memberInner = outer.new MemberInner();
    memberInner.display();
  }
}
```

```
Member Inner Class, count = 5
```

A member inner class is defined as a member of the outer class without `static`. Every member inner object is tied to an outer object. For this reason an outer object is needed when it is created, and the notation is unusual: `outer.new MemberInner()`. Thanks to this tie, `display()` can read the `private`

`count` field of the outer object. Writing `new MemberInner()` directly in `main` does not compile:

```
public class MemberInnerStatic {
  private int count = 5;
  class MemberInner {
    void display() { System.out.println(count); }
  }
  public static void main(String[] args) {
    MemberInner m = new MemberInner();
  }
}
```

```
MemberInnerStatic.java:7: error: non-static variable this cannot be referenced from a
static context
    MemberInner m = new MemberInner();
                    ^
1 error
```

The message again mentions `this`. `new MemberInner()` means `this.new MemberInner()` inside the class: the inner object is to be tied to `this`. Because `main` is `static`, there is no `this`. The trace below shows how the tie is made.

```
public class Outer {
    private int count = 5;
    class MemberInner {
        void display() {
            System.out.println("count = " + count);
        }
    }
    public static void main(String[] args) {
        Outer o1 = new Outer();
        Outer o2 = new Outer();
        o2.count = 8;
        Outer.MemberInner m = o2.new MemberInner();
        m.display();
    }
}
```

```
count = 8
```

Step	Line	Call stack	Variables	New output
1	9: `Outer o1 = new Outer();`	—	o1=→#1	—

Step	Line	Call stack	Variables	New output
2	10: `Outer o2 = new Outer();`	—	o1=→#1, o2=→#2	—
3	11: `o2.count = 8;`	—	o1=→#1, o2=→#2	—
4	12: `Outer.MemberInner m = o2.new MemberInner();`	—	o2=→#2, m=→#3	—
5	13: `m.display();` 4: `void display() {`	—	this=→#3	—
6	5: `System.out.println("count = " + count);`	—	this=→#3, count=Outer.this.count=8	`count = 8`

Table 4.1.18: Tying a member inner object to the outer object.

In the first and second steps two `Outer` objects are created; both have their own `count` field and start with 5. In the third step `o2.count = 8;` changes only object number 2. Because `main` is inside `Outer`, it can write to the `private` field. In the fourth step `o2.new MemberInner()` creates the inner object. The inner object keeps the reference of the outer object that created it in a hidden field. The name of this field in Java is `Outer.this`.

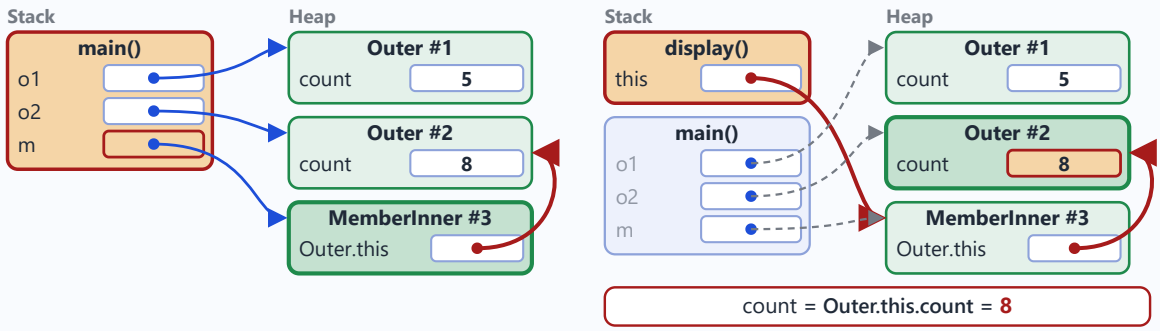


Figure 4.1.20: Left: the inner object (#3) has been created; the hidden `Outer.this` field points to outer object number 2. Right: `display()` is running: `count` means `Outer.this.count`, and the arrow goes to object number 2, to the value 8.

In the fifth step `m.display()` is called. The method's `this` is the inner object. The inner class has no `count` field of its own. In the sixth step the compiler resolves the bare name `count` as `Outer.this.count`: the arrow in the inner object's hidden field goes to outer object number 2, and `count = 8` is printed. Had the inner object been created with `o1.new MemberInner()`, the arrow would go to object number 1 and 5 would be printed. When the inner and outer classes have fields of the same name, the inner class's own field is distinguished by the notation `this.x` and the outer object's field by `Outer.this.x`.

4.1.11.3 Local and anonymous inner classes

```
public class Outer {
  void outerMethod() {
    class LocalInner {         // defined inside the method
      void display() {
        System.out.println("Local Inner Class");
      }
    }
    // usable only inside this method
    LocalInner localInner = new LocalInner();
    localInner.display();
  }

  public static void main(String[] args) {
    Outer outer = new Outer();
    outer.outerMethod();
  }
}
```

```
Local Inner Class
```

A **local inner class** is defined inside a method and can be used only from inside that method. Outside the method the name `LocalInner` is unknown:

```
public class LocalOuter {
  void outerMethod() {
    class LocalInner { void display() { } }
    new LocalInner().display();
  }
  void other() {
    LocalInner li = new LocalInner();
  }
}
```

```
LocalOuter.java:7: error: cannot find symbol
    LocalInner li = new LocalInner();
    ^
  symbol:   class LocalInner
  location: class LocalOuter
LocalOuter.java:7: error: cannot find symbol
    LocalInner li = new LocalInner();
                        ^
  symbol:   class LocalInner
  location: class LocalOuter
2 errors
```

A local inner class is used for a small helper type that is needed several times inside one method. Its scope is like that of a local variable: it ends at the end of the block in which it is defined.

```
interface MyInterface {
  void myMethod();
}

public class Outer {
  public static void main(String[] args) {
    // usually implements an interface or abstract class
    MyInterface myInterface = new MyInterface() {
      @Override
      public void myMethod() {
        System.out.println("Anonymous Inner Class");
      }
    };
    myInterface.myMethod();
    MyInterface short = () -> System.out.println("Lambda");
    short.myMethod();
    System.out.println(myInterface.getClass().getName());
  }
}
```

```
Anonymous Inner Class
Lambda
Outer$1
```

An **anonymous class** is a class without a name; a single object of it is created where it is defined. It usually implements an interface or an abstract class. The notation `new MyInterface() { ... }` means "define a nameless class that implements `MyInterface` and create an object of it". The bodies of the interface's methods are written inside the curly braces. The semicolon at the end belongs to the assignment statement; if it is forgotten, a compile error results.

The nameless class also has a name given by the compiler. The last line of the program prints it: `Outer$1`, that is, the first anonymous class inside `Outer`. A separate file called `Outer$1.class` is also created in the build folder. For interfaces that have only a single abstract method, since Java 8 there is a shorter notation: the lambda expression. `() -> System.out.println("Lambda")` does the same job in one line.

4.1.12 Kinds of classes and design

4.1.12.1 One class, many types

An object can belong to only a single class: it is an object of whichever class it was created from with `new`. But it can have many types. In the definitions below, `Dog` derives from `Animal` and implements the `Runner` interface; `Animal` implements the `Printable` and `Serializable` interfaces.

```
interface Printable { ... }
interface Runner { ... }
class Animal implements Printable, Serializable { ... }
class Dog extends Animal implements Runner { ... }
var winner = new Dog(...);
```

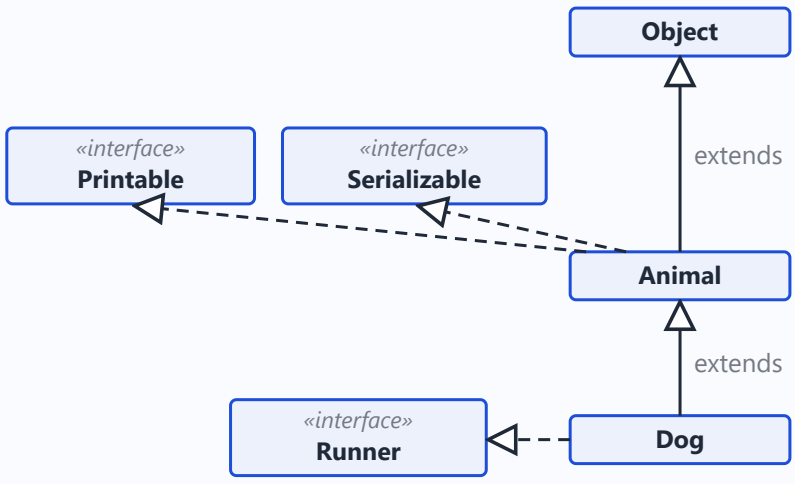


Figure 4.1.21: UML type hierarchy. The solid line with an empty triangle shows inheritance (`extends`), the dashed line shows the implementation of an interface (`implements`). Every class derives from `Object` at the top.

The object the variable `winner` points to belongs only to the class `Dog`. But the types of this object are `Dog`, `Runner`, `Animal`, `Printable`, `Serializable` and `Object`. The `instanceof` operator gives `true` for all six. A small program that fills the definitions with real bodies and tests them:

```
import java.io.Serializable;
interface Printable { }
interface Runner { }
class Animal implements Printable, Serializable { }
class Dog extends Animal implements Runner { }
public class Types {
  public static void main(String[] args) {
    var winner = new Dog();
    Object o = winner;
    System.out.println((o instanceof Dog) + " " + (o instanceof Runner) + " " + (o
instanceof Animal)
        + " " + (o instanceof Printable) + " " + (o instanceof Serializable) + " " + (o
instanceof Object));
    System.out.println(winner.getClass().getName());
  }
}
```

```
true true true true true true
Dog
```

That an object can be seen as more than one type is the basis of polymorphism. A variable of type `Animal` can point to a `Dog` object; a method that expects a `Runner` can take a `Dog`. The conclusion is this: "class" and "type" are not the same concept.

4.1.12.2 Kinds of classes

Classes are written in different forms depending on their purpose. Table 4.1.19 lists the kinds most often met. A class can belong to several kinds at once: `Math` is both a utility class and `final`.

Kind	Property	Example
Enumeration (`enum`)	Its values are a fixed set of objects	`enum Day { MONDAY, TUESDAY, ... }`
Singleton	Only one object can be created	`private` constructor + `static` single object
Abstract (`abstract`)	No object of its own can be created	`abstract class Shape`
Sealed (`sealed` , Java 17+)	Its subtypes are a fixed set	`sealed interface Move permits Walk, Run`
`final`	No subclass can be derived	`final class String`
Record (`record`)	Its objects are immutable	`record Point(double x, double y)`
Utility	Contains only `static` methods	`Math` , `Arrays`
Application	Contains the `main` method	The class where the program starts

Table 4.1.19: Kinds of classes.

```
final class Utils {             // utility, final class
  private Utils() { }         // no instances
  static int square(int x) { return x * x; }
}
class Settings {                   // singleton
  private static final Settings INSTANCE = new Settings();
  private Settings() { }
  static Settings get() { return INSTANCE; }
}
abstract class Shape {            // abstract class
  abstract double area();
}
// sealed (Java 17+)
sealed interface Move permits Walk, Run { }
final class Walk implements Move { }
final class Run implements Move { }
public class Kinds {           // application class
  public static void main(String[] args) {
    System.out.println(Utils.square(7));
    System.out.println(Settings.get() == Settings.get());
  }
}
```

```
49
true
```

`Utils` is a utility class. Because its constructor is `private`, no object of it can be created from outside; no object is needed anyway, because all its methods are `static`. Because it is `final`, no subclass can be derived from it either. The compiler enforces both restrictions:

```
final class Utils {
  private Utils() { }
}
class SubUtils extends Utils { }
public class FinalExtend { }
```

```
FinalExtend.java:4: error: cannot inherit from final Utils
class SubUtils extends Utils { }
                       ^
FinalExtend.java:4: error: Utils() has private access in Utils
class SubUtils extends Utils { }
^
2 errors
```

`Settings` is a singleton class. The single object is created, when the class is loaded, in a `private static final` field. Because the constructor is `private`, no other object can be created; the object is reached only through `get()`. Because `get()` returns the same reference on every call, `Settings.get() == Settings.get()` gives `true`. If `new Settings()` is written in another class:

```
SingletonNew.java:8: error: Settings() has private access in Settings
    Settings a = new Settings();
                 ^
1 error
```

`Shape` is an abstract class: the method `area()` has no body, because the area of a "shape" cannot be computed; only the area of a circle or a square can. An object cannot be created from an abstract class:

```
AbstractNew.java:6: error: Shape is abstract; cannot be instantiated
    Shape s = new Shape();
              ^
1 error
```

`Move` is a sealed interface. The `permits` list names the types that may implement it: only `Walk` and `Run`. If a class that is not in the list tries to implement `Move`, it does not compile:

```
SealedViolation.java:4: error: class is not allowed to extend sealed class: Move (as it is
not listed in its 'permits' clause)
final class Flight implements Move { }
      ^
1 error
```

Sealed types are used where the set of subtypes is closed by design. Because the compiler knows all the possible subtypes, it can check, for example, whether every case of a `switch` has been covered.

4.1.12.3 A design example: Rational

By putting together the pieces so far, a class can be designed from scratch. The `Rational` class represents fractions of the form `num/den`. Three questions are asked when designing a class:

- **Interface**: What will the code that uses the class see? Here a `public` constructor, `add`, which adds two fractions, and `toString`, which gives the fraction as text.
- **Representation**: How will the data be stored? Two `private int` fields: `num` and `den`. The constructor stores the fraction in lowest terms; so every object is always in its simplest form.

- **Implementation**: The bodies of the methods and the helpers not exposed to the outside. The greatest common divisor needed for simplification is found with Euclid's algorithm in a hidden `gcd` method.

```
public class Rational {
  // Representation: hidden fields
  private int num, den;
  // Interface: public constructor and methods
  public Rational(int num, int den) {
    int g = gcd(Math.abs(num), Math.abs(den));
    this.num = num / g;
    this.den = den / g;
  }
  public Rational add(Rational d) {
    int newNum = num * d.den + d.num * den;
    return new Rational(newNum, den * d.den);
  }
  @Override
  public String toString() { return num + "/" + den; }
  // Implementation detail: hidden helper
  private static int gcd(int a, int b) {
    while (b != 0) { int k = a % b; a = b; b = k; }
    return a;
  }
}
```

The test program creates `a = new Rational(8, 10)` and `b = new Rational(2, 3)` and first calls `println(a)`, then `println(a.add(b))`. The whole output of the program is this:

```
4/5
22/15
11/12 3/4
-2/3 1/-3 0/1
1/1
```

The first two lines of the output are these two calls. 8/10 has been simplified and stored as 4/5. 4/5 + 2/3 = (4 × 3 + 2 × 5) / (5 × 3) = 22/15; since 22 and 15 have no common divisor, the fraction stays in lowest terms. The third line is produced by the same call with `a = (6, 8)` and `b = (1, 6)`. This call is traced step by step below.

```
public Rational(int num, int den) {
    int g = gcd(Math.abs(num), Math.abs(den));
    this.num = num / g;
    this.den = den / g;
}
private static int gcd(int a, int b) {
    while (b != 0) { int k = a % b; a = b; b = k; }
    return a;
}
public Rational add(Rational d) {
    int newNum = num * d.den + d.num * den;
    return new Rational(newNum, den * d.den);
}
Rational a = new Rational(6, 8);   // main
Rational b = new Rational(1, 6);
System.out.println(a.add(b) + " " + a);
```

Step	Line	Call stack	Variables	New output

1	14: `Rational a = new Rational(6, 8); // main`
	1: `public Rational(int num, int den) {`
		—	num=6, den=8	—

2	2: `int g = gcd(Math.abs(num), Math.abs(den));` 6: `private static int gcd(int a, int b) {`	—	a=6, b=8	—
3	7: `while (b != 0) { int k = a % b; a = b; b = k; }`	—	a=8, b=6, k=6	—
4	7: `while (b != 0) { int k = a % b; a = b; b = k; }`	—	a=6, b=2, k=2	—
5	7: `while (b != 0) { int k = a % b; a = b; b = k; }`	—	a=2, b=0, k=0	—
6	8: `return a;` 3: `this.num = num / g;` 4: `this.den = den / g;`	—	g=2, this.num=3, this.den=4	—
7	15: `Rational b = new Rational(1, 6);`	—	a=→#1 (3/4), b=→#2 (1/6)	—
8	16: `System.out.println(a.add(b) + " " + a);` 10: `public Rational add(Rational d) {` 11: `int newNum = num * d.den + d.num * den;`	—	this=→#1, d=→#2, newNum=22	—
9	12: `return new Rational(newNum, den * d.den);` 1: `public Rational(int num, int den) {` 2: `int g = gcd(Math.abs(num), Math.abs(den));`	—	num=22, den=24, g=2	—
10	16: `System.out.println(a.add(b) + " " + a);`	—	a=→#1 (3/4), result=→#3 (11/12)	`11/12` `3/4`

Table 4.1.20: Trace of the `Rational` constructor, `gcd` and `add`. The line numbers are the lines of the shortened code above.

In the first step `new Rational(6, 8)` allocates the object; the fields are 0 at first and the constructor's parameters are 6 and 8. In the second step the hidden helper `gcd(6, 8)` is called. Because this method is `private`, the code that uses the class cannot see or call it. It is `static` because it works only with its parameters; it touches no object's fields.

The third, fourth and fifth steps are the three rounds of Euclid's algorithm. In each round `k = a % b` is computed, then `a ← b` and `b ← k`. In the first round 6 % 8 = 6; the numbers just swap places: a = 8, b = 6. In the second round 8 % 6 = 2: a = 6, b = 2. In the third round 6 % 2 = 0: a = 2, b = 0. When `b` becomes zero the loop ends and `a`, that is, 2, is returned. In the sixth step the constructor divides the numerator and denominator by 2 and stores them: the object becomes 3/4.

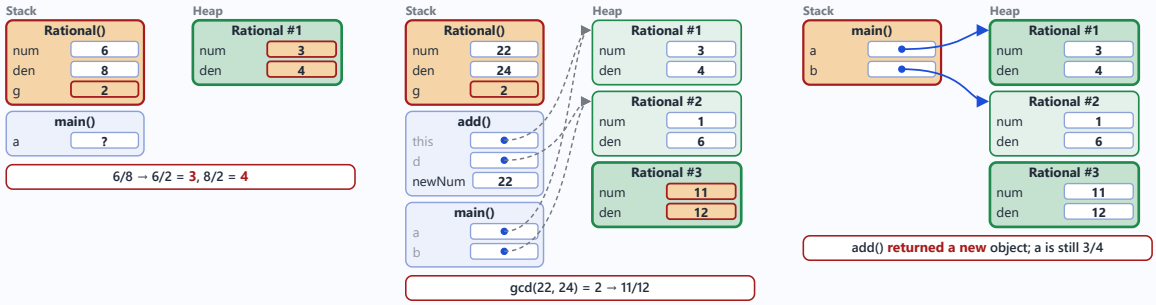


Figure 4.1.22: From left to right: the first object has been simplified, 3/4; a new `Rational(22, 24)` is being built inside `add`; the result 11/12 is a separate object, and `a` is still 3/4.

The seventh step does the same work for `(1, 6)`. `gcd(1, 6)` returns 1; the fraction is already in lowest terms and is stored as 1/6. In the eighth step `a.add(b)` is called. The method's `this` is object number 1 and the parameter `d` is object number 2. `newNum = num * d.den + d.num * den` = 3 × 6 + 1 × 4 = 22. `d.den` is a `private` field of another object, but it can be read because the code is inside the same class.

In the ninth step `new Rational(22, 24)` is called. The constructor runs `gcd` again: `b` takes the values 24, 22, 2, 0 in turn and the result is 2. The new object is stored as 11/12. In the tenth step `println` prints first the returned result and then `a`: `11/12 3/4`. `add` has not changed the existing object; it has returned a new one. `Rational` has no setters and no method changes the fields; its objects are immutable in this sense.

In terms of cost, the loop of `gcd` runs a number of rounds proportional to the number of digits of the smaller number; the number of rounds of Euclid's algorithm is O(log min(a, b)). The constructor and `add` are therefore very fast. Every call of `add` creates a new object; this is the price of the immutable design. In return, even if a variable holding a fraction is shared elsewhere, its value never changes unexpectedly.

The last two lines of the test program try boundary cases. `new Rational(-4, 6)` correctly becomes `-2/3`. For `new Rational(0, 5)`, `gcd(0, 5) = 5` is found and the fraction is stored as `0/1`. On the last line 1/2 + 1/2 = 4/4 is computed and, simplified, becomes `1/1`. These trials expose two limits of the design. `new Rational(3, -9)`, however, is stored as `1/-3`: the constructor does not move the

sign from the denominator to the numerator. Because the same value can be stored in two different forms, an `equals` written for this class might count equal fractions as different. The fix is to flip the signs of both when `den` is negative in the constructor. The second limit is a zero denominator. In the call `new Rational(0, 0)`, `gcd(0, 0)` returns 0 and the division gives this error:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Rational.<init>(RationalZero.java:10)
        at RationalZero.main(RationalZero.java:3)
```

A constructor that does not allow a zero denominator should report the error at the moment the fraction is created and with a meaningful message. For example, the first line of the constructor could be `if (den == 0) throw new IllegalArgumentException("den cannot be 0");`. Keeping the representation hidden makes these fixes easy: the change affects only the constructor, and the code that uses the class stays as it is.

4.1.13 Sample problems and exercises

Example 4.1.2

What does the following program print?

```
class Counter {
  static int total = 0;
  int count = 0;
  void inc() { total++; count++; }
}
public class Problem1 {
  public static void main(String[] args) {
    Counter a = new Counter(), b = new Counter();
    a.inc(); a.inc(); b.inc();
    System.out.print(a.count + " " + b.count);
    System.out.println(" " + Counter.total);
  }
}
```

```
2 1 3
```

`count` is an instance field; every object has a separate copy. Because `a.inc()` is called twice and `b.inc()` once, `a.count` is 2 and `b.count` is 1. `total`, on the other hand, is `static`; all three calls increment the same box and its value becomes 3. The output is `2 1 3`.

Example 4.1.3

What does the following program print?

```
class Box { int value; }
public class Problem2 {
  public static void main(String[] args) {
    Box k1 = new Box();
    k1.value = 5;
    Box k2 = k1;
    k2.value = 9;
    Box k3 = new Box();
    k3.value = 9;
    System.out.println(k1.value);
    System.out.println(k1 == k2);
    System.out.println(k2 == k3);
  }
}
```

```
9
true
false
```

`Box k2 = k1;` creates no new object; `k2` and `k1` point to the same object. `k2.value = 9` writes to this shared object, so `k1.value` is also 9. `k1 == k2` gives `true`. `k3` is a separate object created with `new`; even though its value is also 9, `k2 == k3` gives `false`.

Example 4.1.4

What do the following two programs print?

```
record Money(int lira) { }
public class Problem5 {
  public static void main(String[] args) {
    Money p1 = new Money(10);
    Money p2 = new Money(10);
    Integer i1 = 1000, i2 = 1000;
    System.out.println(p1.equals(p2));
    System.out.println(p1 == p2);
    System.out.println(i1.equals(i2));
    System.out.println(p1);
  }
}
```

```
true
false
true
Money[lira=10]
```

```
class Point {
  int x, y;
  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
public class Problem6 {
  public static void main(String[] args) {
    Point a = new Point(1, 2);
    Point b = new Point(1, 2);
    System.out.println(a.equals(b));
    System.out.println(a.x == b.x);
  }
}
```

```
false
true
```

In the first program `Money` is a record. The record's `equals` compares content: `true`. `==` compares two separate objects: `false`. `i1` and `i2` are two separate `Integer` objects with the value 1000; because they are outside the cache range, `==` would give `false`, but `equals` is always safe: `true`. The last line is the `toString` the record generates. In the second program `Point` is an ordinary class and has not written `equals`. The `Object` version compares references: `false`. `a.x == b.x`, on the other hand, compares two `int` values: `true`.

The following five exercises bring together the topics of these notes. A solution program and its output follow.

1. A `Book` class: `private` `title`, `author`, `pageCount`; a three-parameter constructor, get methods and `toString()`.
2. Add to `Book` a `static` counter `bookCount` and a `setPageCount` that accepts only positive values.
3. A `Circle` class: `radius`, `area()`, `perimeter()`; printing with two decimal places for `new Circle(3)`.
4. Printing the values of the `Season` enum with their ordinal numbers.
5. A `StudentRecord(String name, int id)` record and printing one of its objects.

```
class Book {
  private String title, author;
  private int pageCount;
  private static int bookCount = 0;
  Book(String title, String author, int pageCount) {
    this.title = title;
    this.author = author;
    setPageCount(pageCount);
    bookCount++;
  }
  String getTitle() { return title; }
  String getAuthor() { return author; }
  int getPageCount() { return pageCount; }
  void setPageCount(int pageCount) {
    if (pageCount > 0) this.pageCount = pageCount;
  }
  static int getBookCount() { return bookCount; }
  @Override
  public String toString() { return title + " (" + author + ", " + pageCount + " s.)"; }
}
class Circle {
  private double radius;
  Circle(double radius) { this.radius = radius; }
  double area() { return Math.PI * radius * radius; }
  double perimeter() { return 2 * Math.PI * radius; }
}
enum Season { SPRING, SUMMER, AUTUMN, WINTER }
record StudentRecord(String name, int id) { }
public class Exercises {
  public static void main(String[] args) {
    Book k = new Book("Nutuk", "Atatürk", 600);
    System.out.println(k);
    Book k2 = new Book("Essay", "Author", -5);
    System.out.println(k2.getPageCount() + " " + Book.getBookCount());
    Circle d = new Circle(3);
    System.out.printf("%.2f %.2f%n", d.area(), d.perimeter());
    for (Season m : Season.values()) System.out.print(m.ordinal() + ":" + m + " ");
    System.out.println();
    System.out.println(new StudentRecord("Ali", 7));
  }
}
```

```
Nutuk (Atatürk, 600 p.)
0 2
28,27 18,85
0:SPRING 1:SUMMER 2:AUTUMN 3:WINTER
StudentRecord[name=Ali, id=7]
```

The constructor of `Book` does not write the page count directly to the field; it calls `setPageCount`. In this way the check stays in one place and the constructor uses it too. For the second book with −5 pages the setter rejects the value; the field stays at its default value 0. The counter, however, counts both books: `0 2`. For `Circle`, `printf("%.2f %.2f%n", ...)` is used; on a Turkish locale the decimal separator is a comma: π × 9 ≈ 28,27 and 2 × π × 3 ≈ 18,85. The values of `Season` are traversed with `values()`, and `ordinal()` starts at 0.

4.1.14 Common mistakes

The compiler catches most of the errors related to objects and classes, and its message points directly to the location of the error. Some, however, lead to programs that compile but run wrongly; these are more dangerous.

Common mistake

Calling an instance method from `main` without an object. Message: `non-static method greet() cannot be referenced from a static context`. The call is made through an object (`new Greeter().greet()`); if the method really uses no object's data, it is made `static`.

Common mistake

Writing `void` on a constructor. `public void CtorError(String m)` is not a constructor but an ordinary method. `new CtorError("hello")` is rejected with this message: `constructor CtorError in class CtorError cannot be applied to given types`. A constructor has no return type.

Common mistake

Creating an object without arguments when there is only a parameterized constructor. When a constructor is written, the default constructor is not added; `new Product()` gives the message `cannot be applied to given types`. If needed, a no-argument constructor is written as well.

Common mistake

Forgetting `new`. The line `Product u = Product("Pen");` reads like a method call, and the compiler looks for such a method:

```
ForgotNew.java:7: error: cannot find symbol
    Product u = Product("Pen");
              ^
  symbol:   method Product(String)
  location: class ForgotNew
1 error
```

Common mistake

Accessing a `private` field from outside. Message: `balance has private access in BankAccount`. The field is reached through the class's get and set methods.

Common mistake

Using `this` in a `static` method. Message: `non-static variable this cannot be referenced from a static context`. A `static` method has no object.

Common mistake

Forgetting `this` in a constructor (shadowing). `name = name;` compiles but assigns the parameter to itself; the field stays `null`. The correct form is `this.name = name;`. If the field is made `final`, the compiler catches the error.

Common mistake

Writing `this(...)` after the first line. Message: `call to this must be first statement in constructor`. Delegating to the other constructor must be the first statement of the constructor.

Common mistake

Comparing objects with `==`. `k1 == k2` gives `false` for two separate objects with the same content. Content is compared with `equals()`, including for `String` and the wrapper classes. If `equals` has not been written in one's own class, the `Object` version also behaves like `==`.

Common mistake

Using a `null` reference. A method cannot be called on an object that has not been created or after `null` has been assigned; a `NullPointerException` is thrown. Reference fields that have not been given a value are also `null`. A `null` check is made before use.

Common mistake

Trying to change a record field. Message: `x has private access in Point`. A record is immutable; a new object is created for a new value.

Common mistake

Creating a member inner class directly in `static main`. Message: `non-static variable this cannot be referenced from a static context`. The correct form is `outer.new MemberInner()`; or the inner class is made `static`.

4.1.15 Summary

- **Object-oriented programming** builds a program from interacting objects. Every object has **state** (fields) and **behavior** (methods). The three basic features are encapsulation, inheritance and polymorphism; abstraction is often added.
- A **class** is the blueprint of an object; an **object** is an instance produced with `new`. Objects of the same class carry the same field names, but each has its own values. The method code is single in the class.
- **Access modifiers**: `public`, `protected`, package access, `private`. Fields are `private` and the methods offered to the outside are `public`; the value is read with a getter and written with a setter that performs checks (**data hiding**).
- A **constructor** has the same name as the class and no return type. `new` allocates space on the heap, gives the fields default values, calls the constructor and returns the reference. If no constructor is written, the compiler adds a no-argument **default constructor**; once a constructor is written, it does not.
- Default values of fields: numbers 0, `boolean` `false`, `char` `'\u0000'`, references `null`. Local variables get no default value.
- An **instance field** is separate in every object; a `static` **field** is single and shared in the class. A `static` method has no `this`; it reaches instance members only through an object. A `final` field is given a value once; class constants are `static final` and written in capitals.

- Initialization order: `static` fields and `static { }` blocks once, when the class is first used; then at every `new` the default values, the field initial values and `{ }` blocks (in the order written), and finally the body of the constructor.
- A variable holds not the object but the **reference**. Assignment copies the reference and creates an **alias**. Passing an object to a method also gives a copy of the reference. Access through a `null` reference throws a `NullPointerException`; the **garbage collector** removes an object that has no reference left.
- `this` is the object the method was called on; a shadowed field is reached with `this.name`, and the object can be passed to other methods with `this`. `this(...)` calls another constructor of the same class, must be on the first line and builds a single object.
- `==` tests reference equality and `equals` tests content equality. When `equals` is overridden, `hashCode` is written too. `String` and the wrappers are compared with `equals`.
- **Wrapper classes** carry primitive values as objects. Autoboxing is a call to `valueOf` and unboxing a call to `xxxValue()`. `Integer`s between −128 and 127 come from the cache; unboxing a `null` wrapper throws a `NullPointerException`.
- A **record** is an immutable data class; the constructor, accessors, `equals`, `hashCode` and `toString` are generated automatically. An **enum** is a fixed set of values; it is used with `values()`, `valueOf()`, `ordinal()` and `==`.
- **Inner classes**: `static` (no outer object needed), member (tied to an outer object, `outer.new`), local (inside a method) and anonymous (nameless, a single object).
- An object belongs to a single class, but it can have many types. Kinds of classes: `enum`, singleton, abstract, sealed, `final`, record, utility and application class. In class design the interface, representation and implementation are separated; the representation is kept hidden.

4.1.16 Key terms

object-oriented programming procedural programming class object instance member field method dot operator access modifier getter setter data hiding encapsulation constructor default constructor constructor chaining reference heap stack null reference garbage collector static member initializer block this keyword shadowing wrapper class autoboxing / unboxing record enum immutable inner class anonymous class sealed class / interface abstraction

---

4.1.17 Test Yourself

1. Three objects (`p`, `q`, `r`) are created in a row from the class `class Counter2 { static int s =`

`0; int n; Counter2() { s += 2; n = s; } }`. What are `p.n`, `q.n`, `r.n` and `Counter2.s`?

Answer

`s` is `static` and all three constructors increment the same box: 2, 4, 6. Each constructor copies the current value into the object's own `n` field. A later change of `s` does not affect the `n` of earlier objects. Result: `2 4 6 6`.

2. The class `Cell` has a single `int v` field and a constructor that assigns `v`. The method `static void change(Cell h, Cell g) { h.v = 10; g = h; g.v = 20; }` is called as `change(a, b)` with `a = new Cell(1)` and `b = new Cell(2)`. What are `a.v` and `b.v` afterwards?

Answer

`h` points to the object of `a`; `h.v = 10` makes it 10. `g = h` only points the local copy `g` at the object of `a`; `b` does not change. `g.v = 20` now writes to the object of `a`. Result: `a.v = 20`, `b.v = 2`.

3. Two objects are created from the class `class C { static int t = 0; int a = ++t; { t *= 10; } C() { t++; } }`. What are the `a` of the first object, the `a` of the second object and finally `C.t`?

Answer

For every object the order is: field initial value, then instance initializer, then constructor body. First object: `a = ++t` → t = 1, a = 1; `t *= 10` → 10; `t++` → 11. Second object: a = 12, t = 120, t = 121. Result: `1 12 121`.

4. The class `Chain` has the fields `int x; String s = "";` and these constructors: `Chain() { this(1); s += "a"; }`, `Chain(int x) { this(x, x + 1); s += "b"; }`, `Chain(int x, int y) { this.x = x * y; s += "c"; }`. After `new Chain()`, what are `x` and `s`?

Answer

The chain descends to the innermost constructor: `Chain(1, 2)`. The initial value `s = ""` is given once, immediately before the body of this constructor. The innermost body runs first: x = 2, s = "c". As the chain unwinds, each constructor runs its remaining line: "cb", then "cba". Result: `2 cba`.

5. After the declarations `Integer i1 = 127, i2 = 127, i3 = 1000, i4 = 1000; Long l = 127L;`,

what do `i1 == i2`, `i3 == i4`, `i3.equals(i4)`, `l.equals(127)` and `l == 127` give?

Answer

`true false true false true`. 127 comes from the cache; for 1000 two separate objects are created. `equals` compares values. In `l.equals(127)` the 127 is boxed into an `Integer`; the `equals` of `Long` counts only another `Long` as equal, so the result is `false`. In `l == 127`, `l` is unboxed and the two numbers are compared: `true`.

6. With the `Rational` class in this chapter, how is `new Rational(10, -4)` printed? What does `new Rational(2, 6).add(new Rational(1, 6))` give?

Answer

`gcd(10, 4) = 2`; the numerator and denominator are divided by 2 and the sign stays in the denominator: `5/-2`. In the second expression 2/6 is first stored as 1/3. 1/3 + 1/6 = (1 × 6 + 1 × 3) / 18 = 9/18, and the constructor simplifies it: `1/2`.

7. To get rid of the `non-static method ... cannot be referenced from a static context` error in `main`, a student makes all the fields and methods in the class `static`. The program compiles. What is the problem with this solution?

Answer

When the fields are `static`, a single copy of each field is left in the class. Even if two objects are created, they share the same fields; changing the name of one changes the name of the other. The program effectively becomes procedural. The right solution is to call the object-dependent methods through an object and to make `static` only those that are truly independent of any object.

8. What happens with `new Rational(0, 0)` in the `Rational` class? Which method must be changed, and how, to remove this error and the sign problem? Does the code that uses the class have to change?

Answer

`gcd(0, 0)` returns 0 and the line `num / g` throws `ArithmeticException: / by zero`. Only the constructor changes: at the start, if `den == 0`, an `IllegalArgumentException` with a meaningful message is thrown; after simplification, if `den` is negative, the signs of numerator and denominator are flipped. Because the fields are `private` and the representation is hidden, the code that uses the class does not change at all.

4.1.18 Further reading

- Oracle, *The Java Tutorials*, the "Classes and Objects" lesson: covers defining classes, constructors, `this`, access control, class members and nested classes with short examples; it is free and online.
- A. B. Downey and C. Mayfield, *Think Java* (2nd ed., O'Reilly, 2019), Chapters 10 and 11: explains objects, references and writing one's own class step by step with stack diagrams.
- J. Bloch, *Effective Java* (3rd ed., Addison-Wesley, 2018): an experienced programmer's advice on writing `equals`, `hashCode` and `toString` correctly, immutable classes and `static` factory methods.
- J. Gosling et al., *The Java Language Specification*, Java SE 21, Chapter 8: defines the precise rules for fields, initializers, constructors, `enum`s and records; it is the source to consult for questions about initialization order.

4.1.19 References

- Oracle, *The Java Tutorials*: "Classes and Objects", "Nested Classes", "Enum Types", "Record Classes", "Autoboxing and Unboxing".
- J. Gosling et al., *The Java Language Specification*, Java SE 21 Edition (Chapter 8 Classes: 8.3 Fields, 8.6–8.7 Initializers, 8.8 Constructors, 8.9 Enum Classes, 8.10 Record Classes).
- P. Deitel and H. Deitel, *Java: How to Program, Early Objects*, 11th ed., Pearson, 2017 (Chapters 3 and 8: Classes and Objects).
- K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly, 2022.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., 2022 (Chapter 4: Objects and Classes).
- A. B. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019 (Chapters 10–11).
- J. Bloch, *Effective Java*, 3rd ed., Addison-Wesley, 2018 (the items on `equals`, `hashCode`, `toString`).

PART II · CHAPTER 4.2

Encapsulation

Writing a class begins with bringing data and the methods that work on it together. But this togetherness is not enough by itself. If the fields are open to everyone, every user changes the object as they please; a clock with 75 seconds or an account with a negative balance appears. Encapsulation closes the object's data off from the outside and allows it to be reached only through the class's own methods. The rules are then enforced in a single place, inside the class. These notes build this idea step by step. The first sections explain information hiding, the "what / how" distinction and the contract made of a method's preconditions and postconditions. Access modifiers, getter and setter methods and a detailed trace of a bank account class follow. The notes then separate the interface from the implementation and examine leaked references and immutable classes. The last section covers grouping classes in packages and the `import` declaration. All programs were compiled and run with JDK 21; the outputs, the final values in the trace tables and the compiler messages come from those runs.

LEARNING OUTCOMES

Explain encapsulation

Explain with examples gathering data and the methods that process it in one class, hiding internal details (information hiding) and the "what / how" distinction.

Write a method contract

State a method's precondition and postcondition in a Javadoc comment; choose what to do when the precondition is broken and implement it in code.

Choose access correctly

Say who can reach members at the `public`, `protected`, package and `private` levels; read the access errors the compiler gives.

Write getters and setters

Make fields `private` and write access methods that validate; know the `this.field = field` pattern and the shadowing trap.

Separate interface from implementation
Change a class's internal structure without changing its outside face; meet `interface` and `implements`; close a leaked reference with a defensive copy and design an immutable class.

Use packages
Group classes with `package` and `import`, compile and run a packaged program on the command line, and resolve class name clashes.

4.2.1 Encapsulation and information hiding

Once the fields and methods of a class have been defined, the first design question is this: who may touch these fields? If every field of a class can be read and written from anywhere, the class's own rules stay on paper. The seconds of a clock must be between 0 and 59. The class knows the rule, but every line that changes the field has to remember it separately. If the program has hundreds of lines, forgetting it in one of them is inevitable. Encapsulation solves this problem at the root: it leaves the job of enforcing the rule to the class itself.

4.2.1.1 What is encapsulation?

Definition 4.2.1

Encapsulation is gathering an object's data (its fields) and the methods that process this data in a single unit, the class. The data is closed off from the outside (`private`); from outside, only the methods the class offers can be called.

There are two separate ideas in the definition. The first is **bringing together**: the data and the code that processes it stay in the same class. The second is **closing off**: the data is not directly visible or changeable from outside. Without the second idea the first is half-done. A class with `public` fields also keeps data and methods together, but it does not protect the data.

Encapsulation can be compared to a medicine capsule. The powder inside the capsule is not visible; the patient only swallows the capsule and does not care how the powder was prepared. A class is likewise the guardian of its own data. A call that wants to make the seconds 75 cannot pass through the door of the class. Figure 4.2.1 shows this arrangement on a `DigitalClock` object.

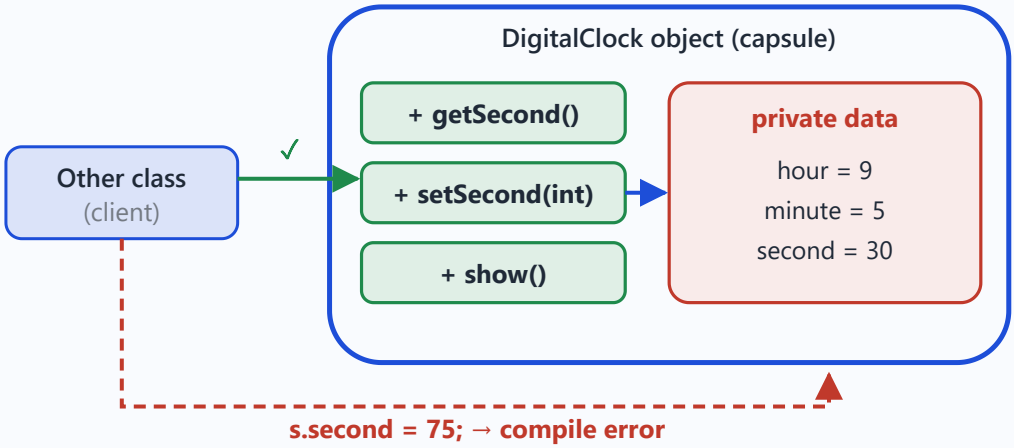


Figure 4.2.1: An encapsulated `DigitalClock` object. The private data (`hour`, `minute`, `second`) is inside the object and closed to the outside. Another class can call only the open methods such as `getSecond()`, `setSecond(int)` and `show()`. An attempt to write to the field directly (`s.second = 75;`) is stopped by a compile error.

The dashed red arrow in the figure shows the most concrete gain of encapsulation. Writing to the field directly is not a run-time error but a **compile error**. The program can never see the wrong value, because such a program does not compile. Stopping the invalid value is not left to run time; the compiler catches it while the code is being written.

Encapsulation is one of the four basic principles of object-oriented programming. The others are inheritance, polymorphism and abstraction. The four complement each other: inheritance and polymorphism organize the relations between classes; encapsulation and abstraction determine how a single class appears to the outside.

4.2.1.2 Information hiding: what and how

Definition 4.2.2

Information hiding is the principle of hiding the internal structure of a class from those who use it. The design separates *what* a task does from *how* it is done. The user knows only what the method is for; the method can be used correctly without knowing the internal details.

Information hiding is a concept intertwined with **abstraction**, and some sources refer to the two by the same name. Abstraction means setting detail aside and showing only the important features. Information hiding makes sure that these details really do not reach the user. One is a point of view, the other is the wall that protects that point of view.

The best-known everyday example is the car. The driver sees and uses the steering wheel, the accelerator pedal and the brake. The driver does not need to know how the steering box, the fuel injection or the brake hydraulics work. Even if the manufacturer changes the engine, the driver goes on driving the car in the same way. Figure 4.2.2 draws this distinction as two layers.

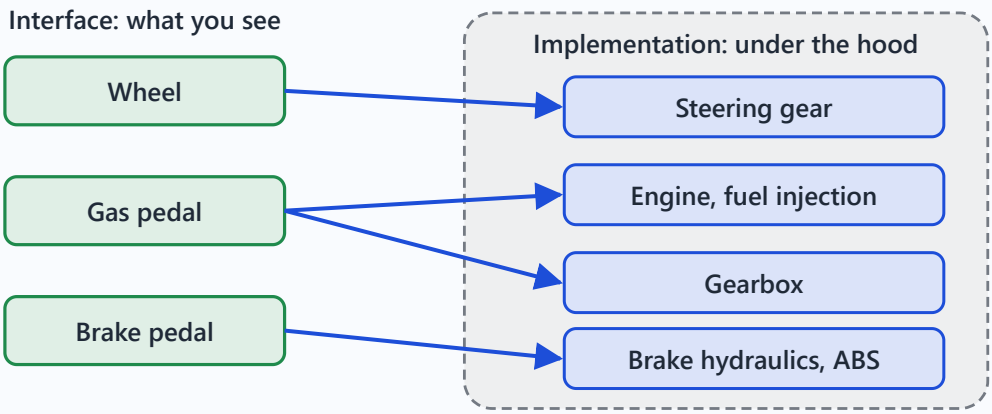


Figure 4.2.2: The car analogy. On the left the interface the driver sees (steering wheel, accelerator pedal, brake pedal), on the right the hidden implementation under the hood (steering box, engine and fuel injection, transmission, brake hydraulics and ABS). Each control activates one or several internal parts the driver knows nothing about.

The same idea is used in Java every day. The `System.out.println` method is called constantly, but there is no need to know how the characters are sent to the screen. Details such as the code that talks to the operating system, character encoding and buffer memory are hidden behind `println`. The only thing the user knows is what the method does: it prints the given value and moves to the end of the line.

4.2.1.3 Behind a single call

To see information hiding in code, consider a small class that adds a record to a database. The `DataService` class gathers the steps of opening the connection, writing the data and closing the connection inside a single method:

```
public class DataService {
    private DbConnection conn;
    // usable without knowing the internals.
    public void addData(String data) {
        conn = new DbConnection();
        conn.connect();
        conn.addData(data);
        conn.close();
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        DataService service = new DataService();
        service.addData("Ayşe");
    }
}
```

To compile this program a `DbConnection` class is also needed. In a real application this class would talk to a database driver. In the example of these notes it is a simple class that only prints to the screen:

```
public class DbConnection {
    public void connect() { System.out.println("Connection opened"); }
    public void addData(String data) { System.out.println("Added: " + data); }
    public void close() { System.out.println("Connection closed"); }
}
```

```
Connection opened
Added: Ayşe
Connection closed
```

The only thing the `Main` class does is the call `service.addData("Ayşe")`. This line says *what* is wanted: "add the record Ayşe." Opening the connection, writing the data and closing the connection, that is, *how* the job is done, is hidden inside `DataService`. Because the `conn` field is `private`, `Main` can neither see nor change it. If tomorrow a real driver were used instead of `DbConnection`, not a single line of `Main` would change.

What happens behind the call is given step by step in Table 4.2.1. The line numbers follow the layout of the trace code in which the two classes are written one after the other: lines 1–10 are the `DataService` class and lines 11–16 the `Main` class.

Step	Line	Call stack	Variables	New output
1	13: `DataService service = new DataService();`	main()	service=→DataService	—
2	4: `public void addData(String data) {`	main() → service.addData("Ayşe")	data="Ayşe"	—
3	5: `conn = new DbConnection();`	main() → service.addData("Ayşe")	data="Ayşe", conn=→object	—
4	6: `conn.connect();`	main() → service.addData("Ayşe") → conn.connect()	data="Ayşe"	`Connection opened`
5	7: `conn.addData(data);`	main() → service.addData("Ayşe") → conn.addData(data)	data="Ayşe"	`Added: Ayşe`

Step	Line	Call stack	Variables	New output
6	8: `conn.close();`	main() → service.addData("Ayşe") → conn.close()	data="Ayşe"	`Connection closed`
7	15: `}`	main()	service=→DataService	—

Table 4.2.1: Trace of the call `service.addData("Ayşe")`. In the call stack column the rightmost frame is the topmost; the variables column shows the frame running at that moment.

In the first step `new DataService()` creates an object on the heap. The `service` variable is a reference to this object. The object's `conn` field is still `null`, because the default value of fields of reference type is `null`. In the second step `addData` is called; a new frame is added to the stack and the `data` parameter receives the value `"Ayşe"`.

In the third step the hidden work begins. A second object is created and the `private` `conn` field points to it. Figure 4.2.3 gives this moment and the end of the call side by side. In the fourth step `connect()` is called; there are now three frames on the stack: `main`, `addData` and `connect`. `Main` is aware of none of these frames. In the fifth step the data goes to the `addData` method of the connection object. The two methods have the same name but belong to different classes; Java knows which one is called from the type of the object. In the sixth step the connection is closed.

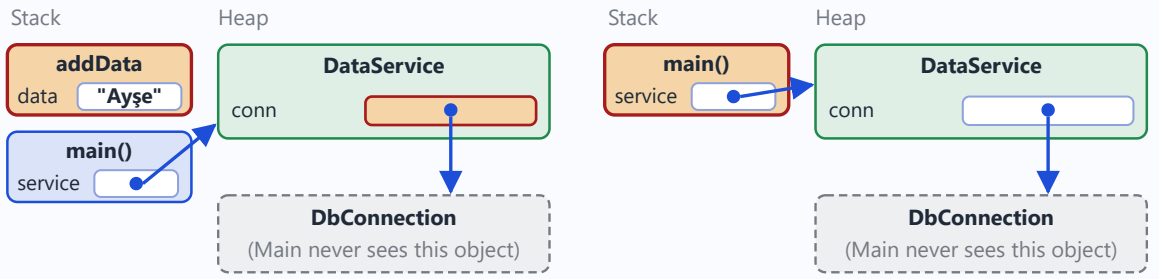


Figure 4.2.3: Two moments of a single call. Left: `addData` is running; the private `conn` field of the `DataService` object points to the newly created `DbConnection` object. Right: the call has ended and only `main` is left on the stack; the connection object sits in a place the user never sees.

In the seventh step the flow returns to `main`. The user wrote one line and knew none of the three separate operations. This is exactly what information hiding means. This arrangement has no cost: every frame added to the call stack is opened and closed in constant time. The hidden work is the same work that would have to be done if it were not hidden.

Common mistake

Writing code that uses the hidden detail outside. A programmer who opens and closes the connection in `Main` does the work of `DataService` a second time. When the form of the connection changes, two places have to be fixed. Where a class offers a task with a single call,

the steps of that task are not rewritten outside.

4.2.2 Preconditions and postconditions

Information hiding asks that the user need not know the inside of the method. But the user must know one thing: in what situation does the method work correctly, and what does it guarantee when it does? This information is not inside the code but on the outer face of the method. This section describes the agreement a method makes with its user and how this agreement is enforced in code.

4.2.2.1 The contract of a method

Definition 4.2.3

A **precondition** is a condition assumed to be true before the method is called; ensuring it is the caller's responsibility. A **postcondition** is the state guaranteed to be true after the method finishes, provided the precondition was met; ensuring it is the method's responsibility. Together they make up the method's **contract**.

The contract has two sides. The caller says "I am giving non-empty data", and the method answers "And I add it and return `true`." The user knows only the contract, not the body. If the precondition is broken, the method does not have to guarantee anything. This means responsibility is shared openly: if the fault is in the value the caller gave, the caller fixes it; if the method gives a wrong result with a correct value, the method is fixed.

A precondition and a postcondition can be written as comment lines as a first step. The two small pieces in the slides state first the precondition and then the postcondition of the same `addData` method:

```
public class DataService {
    // Precondition: data != null
    public void addData(String data) {
        // check whether data is null
        // ...
    }
}
```

```
public class DataService {
    // Postcondition: returns true if data was added
    public boolean addData(String data) {
        // check whether the data was added
        // ...
        return true;
    }
}
```

In the second piece the return type of the method has also changed: `boolean` instead of `void`. Because the postcondition says "returns `true` if added", the method has to return a value. Writing the contract forces one to think about the method's signature as well.

4.2.2.2 Enforcing the contract in code

A comment line checks nothing by itself. The version below both writes the contract in the comment and enforces it on the first line of the body:

```
public class DataService {
  private int count = 0;

  /**
   * Adds a record to the database.
   * Precondition: data is not null or empty.
   * Postcondition: returns true if added,
   *   and count grows by 1.
   */
  public boolean addData(String data) {
    if (data == null || data.isEmpty()) {
      return false;   // precondition broken
    }
    // ... add data to the database ...
    count++;
    return true;      // postcondition met
  }

  public int getCount() {
    return count;
  }
}
```

The `/** ... */` at the top of the method is a **Javadoc** comment: it explains how the method is to be used and can be turned into documentation with tools. The first job of the body is to check the precondition. If `data` is `null` or empty, the method returns `false` without touching the object. If the precondition holds, the record is added, `count` goes up by one and `true` is returned; the postcondition is thus fulfilled. The first `main` method in the slides makes three calls:

```
// in main:
DataService service = new DataService();
System.out.println(service.addData("Ali"));
System.out.println(service.addData(""));
System.out.println(service.addData(null));
System.out.println(service.getCount());
```

```
true
false
false
1
```

The trace of four calls is given in Table 4.2.2. This time `"Ayşe"`, `null`, `""` and `"Can"` are tried in turn. The line numbers follow the layout of the trace code: lines 1–7 are `addData` and lines 9–14 are in `main`.

Step	Line	Variables	New output
1	9: `DataService service = new DataService();`	service=→#1, count=0	—
2	10: `System.out.println(service.addData("Ayşe"));` 2: `if (data == null || data.isEmpty()) {`	data="Ayşe", count=0	—
3	5: `count++;` 6: `return true; // postcondition met`	data="Ayşe", count=1	`true`
4	11: `System.out.println(service.addData(null));` 2: `if (data == null || data.isEmpty()) {`	data=null, count=1	—
5	3: `return false; // precondition broken`	data=null, count=1	`false`
6	12: `System.out.println(service.addData(""));` 2: `if (data == null || data.isEmpty()) {` 3: `return false; // precondition broken`	data="", count=1	`false`
7	13: `System.out.println(service.addData("Can"));` 5: `count++;` 6: `return true; // postcondition met`	data="Can", count=2	`true`
8	14: `System.out.println(service.getCount());`	count=2	`2`

Table 4.2.2: Trace of the `addData` contract with four calls. `count` in the variables column is the object's field.

In the second step `"Ayşe"` is neither `null` nor empty. Both sides of the `||` operator are false, and the `if` block is skipped. In the third step the counter becomes 1 and `true` is returned. The fourth step is the most instructive: `data == null` is true and `||` **short-circuits**. Because the left side is true, the right side is never evaluated; the call `null.isEmpty()` is not made and no `NullPointerException` occurs. Figure 4.2.4 shows this case side by side with the empty-string case.

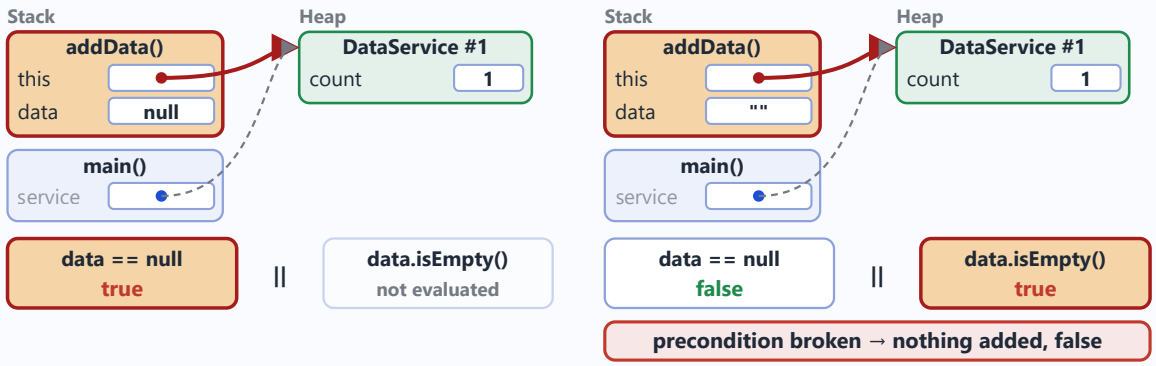


Figure 4.2.4: The precondition broken in two different ways. Left: `data` is `null`; `data == null` is true and `data.isEmpty()` is never evaluated. Right: `data` is the empty string; the first condition is false and the second is true. In both cases `count` stays at 1.

This is why the order of the conditions matters. Had it been written as `data.isEmpty() || data == null`, `isEmpty()` would be called first on a `null` argument and the program would stop with a `NullPointerException`. When tried with JDK 21 this is indeed what happens. First the `null` check, then the call on the object: this order is always kept in Java.

In the sixth step the empty string arrives. `""` is an object, not `null`; but because its length is 0, `isEmpty()` returns true. In the seventh step `"Can"` is added and the counter becomes 2. Of the four calls, only the two that kept the contract added a record. The state of the object was never corrupted in any call.

One more detail: `" "` (a single space) passes the `isEmpty()` test, because its length is 1. When `addData(" ")` is sent to the same object, the method returns `true` and counts the record. If data made up of spaces should also be rejected, the condition is written with `data.trim().isEmpty()` or, since Java 11, `data.isBlank()`. What exactly the precondition says determines what is checked in the code.

The cost of checking the precondition is small: two comparisons and at most one method call, that is, constant time. Its gain, on the other hand, is large; an invalid record can never get into the object.

4.2.2.3 When the precondition is broken

What the method does when the precondition is broken is also a design decision. Table 4.2.3 compares three common approaches.

Approach	Example	What does the caller notice?
Silently ignore	`setSecond(75)` does nothing	Nothing; the error may go unnoticed.
Return a status	`addData(null)` → `false`	Notices only if it looks at the return value.
Throw an exception	`throw new IllegalArgumentException(...)`	If not caught, the program stops; the error is not lost.

Table 4.2.3: Three approaches to choose from when the precondition is broken.

The first approach is the simplest, but it hides the error. The caller thinks it wrote 75, yet the clock has not changed. The second approach reports the error, but it depends on the caller looking at the return value. A call that ignores the return value is as silent as in the first approach. The third approach makes the error impossible to ignore. The `setSecond` below throws an `IllegalArgumentException` for an invalid value:

```
public void setSecond(int second) {
    if (second < 0 || second > 59)
        throw new IllegalArgumentException(
            "second must be 0-59: " + second);
    this.second = second;
}
// in main:
DigitalClock s = new DigitalClock();
try {
    s.setSecond(30);
    s.setSecond(75);
    System.out.println("never reached");
} catch (IllegalArgumentException e) {
    System.out.println("Error: " + e.getMessage());
}
System.out.println(s.getSecond());
```

```
Error: second must be 0-59: 75
30
```

`s.setSecond(30)` succeeds and the seconds become 30. In the call `s.setSecond(75)` the condition is true and the `throw` statement throws the exception. The assignment line does not run; the method ends at that point. The flow skips the rest of the `try` block, so `"never reached"` is not printed. Control passes to the `catch` block and the exception's message is printed. On the last line the seconds are still 30; the state of the object has not been corrupted.

Definition 4.2.4

An **exception** is an event that disrupts the normal flow while the program is running and is thrown as an object. The `throw` statement throws an exception object; an exception thrown in a `try` block is caught in the `catch` block that matches its type.

What happens if the exception is not caught? When the same `setSecond(75)` call is made outside a `try`, JDK 21 gives this output:

```
Exception in thread "main" java.lang.IllegalArgumentException: second must be 0-59: 75
        at DigitalClock.setSecond(DigitalClock.java:6)
        at Uncaught.main(Uncaught.java:4)
```

The first line of the message shows the type and message of the exception, and the following lines show where the exception was born. `DigitalClock.java:6` is the line where the `throw` statement is; `Uncaught.java:4` is the line where the faulty call was made. This is the strength of the exception: the faulty value becomes visible at the place where the call was made.

For argument errors in `public` methods, Java's customary choice is `IllegalArgumentException`. The Java library follows the same path. For example, `Integer.parseInt("abc")` throws a `NumberFormatException`, and this class is a subtype of `IllegalArgumentException`. Whichever approach is chosen, the choice must be documented as part of the contract. The user must be able to read from the documentation what the method will do with a bad argument.

4.2.2.4 Writing the contract with Javadoc

Javadoc comments make it possible to write the contract in a form that is both readable and processable by tools. The comment starts with `/**`; the first sentence summarizes what the method does. The **tags** that follow state the parts of the contract separately:

```
/**
 * Withdraws money from the account.
 * @param amount sum to withdraw; {@code 0 < amount <= balance}
 * @return the balance after the call (old balance - amount)
 * @throws IllegalArgumentException if amount is out of range
 */
public double withdraw(double amount) {
    if (amount <= 0 || amount > balance)
        throw new IllegalArgumentException("Invalid amount: " + amount);
    balance -= amount;
    return balance;
}
```

Tag	What is written?	In the example
`@param`	Each parameter and its precondition	for `amount`, `0 < amount <= balance`
`@return`	The return value, that is, the postcondition	the balance after the call
`@throws`	What happens when the contract is broken	`IllegalArgumentException`

Table 4.2.4: Javadoc tags used to write the contract.

The `{@code ...}` notation in the comment is not a mere detail. Javadoc processes the inside of comments as HTML. A raw `<` sign is taken to be the start of an HTML tag. When the same comment is written as `0 < amount <= balance` without `{@code}`, the `javadoc` tool of JDK 21 gives two

`malformed HTML` errors, and in the generated page an `invalid input` warning appears in place of the inequality. The text inside `{@code ...}`, on the other hand, is shown as it is, in a code font.

The command `javadoc -d docs BankAccount.java` produces HTML pages in the `docs` folder from these comments. The online documentation of the Java library is produced the same way. Every explanation read in the documentation of the `String` or `Math` class comes from the Javadoc comments in the source code of those classes.

The answer to the question of what to write in Javadoc comes from information hiding. What the method does and its conditions of use are written. **How** it does it is not written; that is the hidden implementation and it may change. The reader of Javadoc is the user who will never see the body. From this point of view, the outer face of a class is the sum of the signatures of the `public` methods and these comments.

The `withdraw` in the example enforces the contract with an exception. When tried with JDK 21 on an account with a balance of 1000 TL, `withdraw(300.0)` returns 700.0. `withdraw(5000.0)` and `withdraw(-20)` throw the exception with the messages `Invalid amount: 5000.0` and `Invalid amount: -20.0`. Note the `-20.0` in the second message: the argument `-20` given as an `int` has been widened to the `double` parameter.

4.2.2.5 Class invariants and assert

A precondition and a postcondition belong to a single method. Some rules, however, must hold at every moment for all objects of the class. The seconds of a clock must always be between 0 and 59, and the balance of a bank account must always be zero or positive.

> **Definition 4.2.5**
>
> A **class invariant** is a rule that the state of every object of a class must satisfy at all times. The invariant is established when the object is constructed and must be true again when each `public` method finishes.

To check an invariant during development, Java's `assert` statement is used. The statement `assert condition : message;` evaluates the condition; if the condition is false, it throws an `AssertionError` with the message. In the `Counter` class below an error has been left in on purpose: the value becomes 10 after 9, although the invariant requires the range 0–9.

```
class Counter {
    private int value;   // invariant: 0 <= value <= 9
    void increment() {
        value = value + 1;   // bug: 10 after 9
        assert value >= 0 && value <= 9 : "value=" + value;
    }
}
public class Main {
    public static void main(String[] args) {
        Counter s = new Counter();
        for (int i = 0; i < 10; i++) s.increment();
        System.out.println("done");
    }
}
```

The program is run twice. First in the ordinary way, with the command `java Main`:

```
done
```

Then with assertions enabled, with the command `java -ea Main`:

```
Exception in thread "main" java.lang.AssertionError: value=10
        at Counter.increment(Main.java:5)
        at Main.main(Main.java:11)
```

The difference between the two outputs shows the most important property of `assert`. Assertions are **disabled** by default. If the `-ea` (enable assertions) option is not given, the `assert` line does not run at all; the condition is not even evaluated. The error passes silently and the program prints `done`. With `-ea`, on the tenth increment the value becomes 10, the condition comes out false and `AssertionError: value=10` is thrown. The first line of the stack trace shows that the error was born in `Counter.increment`, on line 5 of the file `Main.java`.

This property also determines what `assert` is used for. `assert` checks, during development, internal assumptions about which the programmer says "this must never happen": class invariants, postconditions, code branches that should not be reached. The arguments of a `public` method, however, are not checked with `assert`. When assertions are off, no check is made and a faulty argument gets into the object. In short: an exception for arguments, `assert` for internal consistency.

Common mistake

Checking user input with `assert`. The line `assert age >= 0;` works only in a program run with `-ea`. When the program is run in the ordinary way, a negative age is accepted without any warning. For a value coming from outside, a check with `if` and

`IllegalArgumentException` are used.

Common mistake

Writing an expression with side effects in an `assert` condition. A line such as `assert list.remove(x);` does not run at all when assertions are off; the element is not removed from the list. The behavior of the program changes with the `-ea` option. An `assert` condition only reads and changes nothing.

4.2.3 Access modifiers

The "closing off" half of encapsulation is provided in Java by access modifiers. Who may use a field or method is determined by a single keyword written at the start of its declaration. The compiler enforces the rule; it is not left to the programmer's good intentions.

4.2.3.1 public and private

Definition 4.2.6

An **access modifier** is a keyword that determines who can access a class, field or method. A `public` member can be accessed directly by other classes. A `private` member can be accessed only from inside the class; the compiler prevents access from outside.

The habit followed when writing a class can be summarized in three points. Classes are mostly declared `public`. The fields of the class are declared `private`. `public` methods are written to access and change the fields. In short: fields hidden, behaviors open.

```
public class Student {      // public class
    private String name;      // hidden field
    private int grade;        // hidden field

    public int getGrade() {   // public method
        return grade;
    }
}
```

In the `Student` class the fields `name` and `grade` are `private`. Another class cannot read the grade by writing `o.grade`; to read it, `o.getGrade()` is called. Figure 4.2.5 compares these two ways in UML notation. In UML class diagrams the sign `-` means a `private` member and `+` a `public` member.

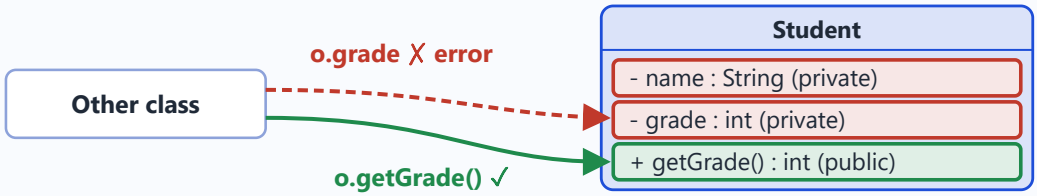


Figure 4.2.5: Another class can call the `public` method `getGrade()` of `Student` (green arrow). An attempt to access the `private` field `grade` directly does not compile (red dashed arrow).

4.2.3.2 The four access levels

Java has two further levels between `public` and `private`. Who each of the four levels is open to is given in Table 4.2.5.

Modifier	Same class	Same package	Subclass (other package)	Everywhere
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	✗
(none) package access	✓	✓	✗	✗
`private`	✓	✗	✗	✗

Table 4.2.5: The four access levels. ✓ means accessible, ✗ means not accessible.

If no modifier is written, **package access** (package-private, called "default" in some sources) applies: only classes in the same package reach the member. `protected` is for subclasses. In addition to classes in the same package, subclasses in other packages also reach a `protected` member. A package is a group in which related classes are collected in the same folder. A subclass is a class derived from another class with `extends`. The levels form nested rings (Figure 4.2.6).

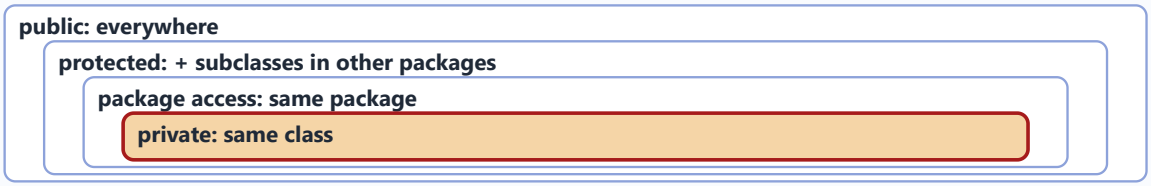


Figure 4.2.6: The access levels are nested regions. In the center is `private` (only the same class), and in the ring around it package access (the same package). Outside that is `protected` (plus subclasses in other packages), and at the outermost `public` (everywhere).

Every cell of the table has been tested with JDK 21. Four fields have been put in class `A` in package `p1`: `public int pub`, `protected int prot`, `int pkg` with no modifier, and `private int priv`. When an attempt is made to write to the four fields from four different places, the compiler gives these errors:

```
p1/B.java:3: error: priv has private access in A
    void tryIt(A a) { a.pub = a.prot = a.pkg = 1; a.priv = 1; }
                                                       ^
p2/C.java:3: error: pkg is not public in A; cannot be accessed from outside package
    void tryIt() { pub = prot = 1; pkg = 1; priv = 1; }
                                    ^
p2/C.java:3: error: priv has private access in A
    void tryIt() { pub = prot = 1; pkg = 1; priv = 1; }
                                           ^
p2/D.java:3: error: prot has protected access in A
    void tryIt(p1.A a) { a.pub = 1; a.prot = 1; a.pkg = 1; a.priv = 1; }
                                     ^
p2/D.java:3: error: pkg is not public in A; cannot be accessed from outside package
    void tryIt(p1.A a) { a.pub = 1; a.prot = 1; a.pkg = 1; a.priv = 1; }
                                             ^
p2/D.java:3: error: priv has private access in A
    void tryIt(p1.A a) { a.pub = 1; a.prot = 1; a.pkg = 1; a.priv = 1; }
                                                         ^
6 errors
```

`A`'s own method can write to all four fields; there is no error at all. `B`, in the same package, cannot reach only `priv`. The subclass `C` in another package cannot reach `pkg` and `priv`. The unrelated class `D` in another package can write only to `pub`. This result is exactly the same as the rows of the table. The messages also vary with the level: `has private access`, `is not public in A; cannot be accessed from outside package` and `has protected access`.

The rule for choosing the right level is simple: prefer the most restrictive level. First think of `private`; move to a more open level only if really needed. Opening a member later is easy and affects no one. Closing an open member later, on the other hand, breaks all the code that uses it.

A last rule concerns the classes themselves. A top-level class, that is, a class that is not inside another class, can be only `public` or package-private. When `private class TopPrivate { }` is written, the compiler gives the message `modifier private not allowed here`. `private` and `protected` classes can be defined only inside another class, as inner classes.

4.2.3.3 Trying to access a private field

In the `DigitalClock` class the `second` field is `private`. What happens if another class tries to write to this field directly?

```
public class Test {
    public static void main(String[] args) {
        DigitalClock s = new DigitalClock();
        s.second = 75;
    }
}
```

```
Test.java:4: error: second has private access in DigitalClock
        s.second = 75;
         ^
1 error
```

The message shows the file and line of the error (`Test.java:4`), its cause (`second has private access in DigitalClock`) and its position within the line (the `^` sign). The program never runs; no `.class` file is produced. Had the field been `public` , `s.second = 75;` would have been accepted silently and the clock would have fallen into an invalid state. The solution is to have the change made by the class's own method: `s.setSecond(75);` . The method rejects the invalid value (Figure 4.2.7).

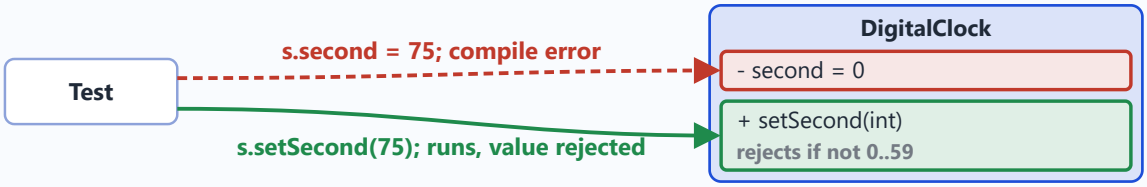


Figure 4.2.7: The two ways of the `Test` class to `DigitalClock` . Writing to the field directly (`s.second = 75;`) is a compile error. Going through the setter (`s.setSecond(75);`) compiles and runs; but a value outside 0–59 is rejected and the field stays at 0.

The difference between the two ways is where the rule is checked. In the first way the compiler does the check and forbids access altogether. In the second way access is allowed, but the value goes through the class's own rule. Encapsulation uses both together: the direct way to the field is closed, and the checked way through the method is open.

4.2.3.4 private works at class level

The word `private` does not mean "private to this object". Access control is done not per object but **per class**. The code of a class can read the `private` fields of *every* object created from that class.

```
class Ratio {
    private int num, den;
    Ratio(int num, int den) {
        this.num = num;
        this.den = den;
    }
    Ratio times(Ratio other) {
        // other's private fields: allowed
        return new Ratio(num * other.num,
                         den * other.den);
    }
    @Override public String toString() {
        return num + "/" + den;
    }
}
// in main:
Ratio k = new Ratio(2, 3).times(new Ratio(3, 4));
System.out.println(k);
```

```
6/12
```

The `times` method reads its own fields (`num` , `den`) and the fields of the `other` object that comes in as a parameter (`other.num` , `other.den`). The second object is another object, but of the same class; so access is free. The product of 2/3 and 3/4 is 6/12. The fraction is not simplified, because `times` only multiplies the numerators and denominators.

Thanks to this, methods that process two objects together, such as `times` , `equals` and `compareTo` , need no separate method to read a field. When the same line is written in another class, however, it does not compile:

```
Outside.java:3: error: num has private access in Ratio
        int p = new Ratio(1, 2).num;
                                ^
1 error
```

The limit of encapsulation is the source code of the class. The inside of the class is the responsibility of a single author, and that author knows all the rules of the class. What is protected is the users outside the class.

4.2.4 Getter and setter methods

Once the fields are `private`, the only way to reach them is through the methods of the class. The two most common kinds of these methods are those that read the value of a field and those that change a field in a checked way.

> **Definition 4.2.7**
>
> An **accessor** (getter) is a method that returns the value of a field. A **mutator** (setter) is a method that first checks a new value for a field and assigns it if it is valid. In Java convention their names start with `get` and `set`.

4.2.4.1 The DigitalClock class

The main example of the slides is a digital clock that keeps the hour, minute and second. The first version contains a getter and a setter only for the second:

```
public class DigitalClock {
    private int hour = 0;
    private int minute = 0;
    private int second = 0;

    public int getSecond() {
        return this.second;
    }

    public void setSecond(int second) {
        if (second >= 0 && second < 60) {
            this.second = second;
        }
    }
}
```

`getSecond()` returns the value of the field; it changes nothing. `setSecond(int second)` first checks the incoming value. If the value is between 0 and 59, the assignment is made; otherwise the method ends without doing anything. This setter has chosen the approach of silently ignoring. The UML diagram of the class is given in Figure 4.2.8.

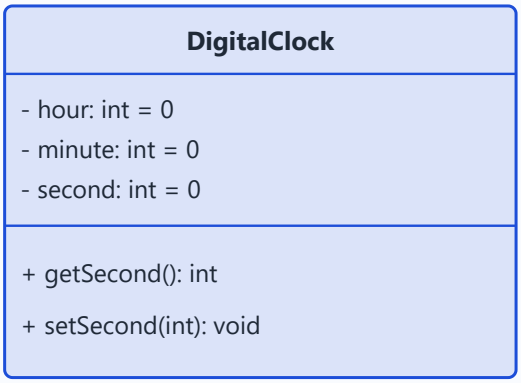


Figure 4.2.8: The UML diagram of the `DigitalClock` class. At the top the name of the class, in the middle the three `private` fields (`-`) and their initial values, at the bottom the two `public` methods (`+`) and their return types.

4.2.4.2 Tracing the setter's protection

To see how the setter stops an invalid value, the program below is traced step by step:

```
public class DigitalClock {
  private int second = 0;
  public int getSecond() {
    return this.second;
  }
  public void setSecond(int second) {
    if (second >= 0 && second < 60) {
      this.second = second;
    }
  }
}
// in ClockTest.main:
DigitalClock s = new DigitalClock();
s.setSecond(45);
s.setSecond(75);
System.out.println(s.getSecond());
```

```
45
```

Step	Line	Call stack	Variables	New output
1	13: `DigitalClock s = new DigitalClock();`	main()	s=→DigitalClock	—
2	7: `if (second >= 0 && second < 60) {`	main() → s.setSecond(45)	this=→object, second=45	—

Step	Line	Call stack	Variables	New output
3	8: `this.second = second;`	main() → s.setSecond(45)	this=→object, second=45	—
4	7: `if (second >= 0 && second < 60) {`	main() → s.setSecond(75)	this=→object, second=75	—
5	4: `return this.second;`	main() → s.getSecond()	this=→object, return=45	—
6	16: `System.out.println(s.getSecond());`	main()	s=→DigitalClock	`45`

Table 4.2.6: Trace of the `setSecond` calls. Line numbers: 1–11 the class, 13–16 the calls in `main`.

In the first step the object is created and the `second` field starts at 0. In the second step `setSecond(45)` is called. There are two things in the new frame: the `second` parameter is 45 and `this` points to the object that called the method. The condition is true. In the third step `this.second = second;` runs. The `this.second` on the left is the object's field, the `second` on the right is the parameter; the field becomes 45.

The fourth step is the protection itself. In the call `setSecond(75)` the condition is false and the assignment line is skipped. The invalid value cannot enter the object. Figure 4.2.9 shows these two calls side by side. In the fifth step `getSecond()` returns the value of the field: 45. In the last step 45 is printed to the screen. The state of the object stayed valid from beginning to end.

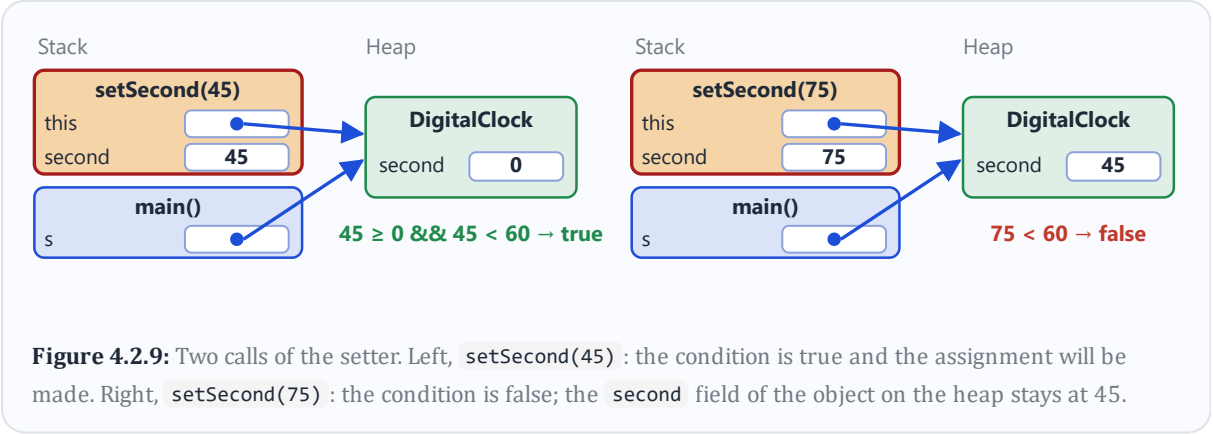


Figure 4.2.9: Two calls of the setter. Left, `setSecond(45)`: the condition is true and the assignment will be made. Right, `setSecond(75)`: the condition is false; the `second` field of the object on the heap stays at 45.

The cost of a getter and a setter is constant: one comparison, one assignment or one read. The Java virtual machine often inlines methods this small at run time; no measurable speed difference remains compared with direct field access. Encapsulation should not be seen as a cost that slows the program down.

4.2.4.3 this and shadowing

In setters the name of the parameter is usually chosen to be the same as the name of the field: `setSecond(int second)`. This choice improves readability, but it requires knowing a rule.

Definition 4.2.8

Shadowing is a name in an inner scope hiding the same name in an outer scope. If the name of a parameter is the same as that of a field, inside the method that name refers to the parameter; the field is left in the shade. `this` is a reference to the object that called the method, and `this.second` always refers to the field.

The correct form is `this.second = second;`. The `Box` class shows what happens when `this` is forgotten:

```
public class Box {
    private int width;
    public int getWidth() {
        return width;
    }
    public void setWidth(int width) {
        width = width;
    }
}
```

In the `setWidth` method both sides of the line `width = width;` are the parameter. The parameter is assigned to itself and the field never changes. After the call `k.setWidth(5)`, `k.getWidth()` returns:

```
0
```

The field has stayed at 0, the default value of `int` fields. The most dangerous side of this error is that the compiler gives no error. `width = width;` is a valid Java statement. The error is noticed only when the program is run and the output is looked at. That is why the habit of writing `this.` in setters is important. Some development environments flag this line with a "self-assignment" warning, but `javac` with default settings stays silent.

Common mistake

Forgetting `this` with a `final` field. If the field is `private final String name;` and `name = name;` is written in the constructor, the result is different. The assignment is again made to the parameter; the field has been given no value. Because a `final` field must definitely be assigned in the constructor, the compiler this time gives the error `variable name might not have been initialized`, pointing at the end of the constructor. With a non-`final` field no warning appears at all.

4.2.4.4 Access methods and naming

When a getter and a setter are written for all the fields of the clock, the class takes this form:

```
public class DigitalClock {
  private int hour = 0;      // 0..23
  private int minute = 0;    // 0..59
  private int second = 0;    // 0..59

  public int getHour() { return hour; }
  public int getMinute() { return minute; }
  public int getSecond() { return second; }

  public void setHour(int hour) {
    if (hour >= 0 && hour < 24) this.hour = hour;
  }
  public void setMinute(int minute) {
    if (minute >= 0 && minute < 60) this.minute = minute;
  }
  public void setSecond(int second) {
    if (second >= 0 && second < 60) this.second = second;
  }

  public void show() {
    System.out.printf("%02d:%02d:%02d%n",
        hour, minute, second);
  }
}
```

```
DigitalClock s = new DigitalClock();
s.setHour(9);
s.setMinute(5);
s.setSecond(30);
s.show();
s.setHour(24);     // rejected
s.setMinute(-3);   // rejected
s.show();
```

```
09:05:30
09:05:30
```

The first three calls are valid and the clock becomes 09:05:30. The call `setHour(24)` is rejected, because valid hours are 0–23. `setMinute(-3)` is rejected too. The second `show()` call therefore prints the same value. The `%02d` format in `printf` writes the number with at least two digits, adding a leading zero if needed: the value 9 appears as `09`.

The names of access methods follow a convention. The getter of the `minute` field is called `getMinute()` and its setter `setMinute(int)`. The first letter of the field name is capitalized and `get` or `set` is put in front. The getter of a `boolean` field, however, starts with `is`: `isActive()` for a field `active`. This convention is not only for readability. Many Java libraries and development tools derive the properties of a class from these names.

4.2.4.5 Why not a public field?

Making a field `private` and writing two methods takes more effort than making it `public`. What this effort buys is summarized in Table 4.2.7.

```
public class Clock {
    public int second;
}
// in another class:
s.second = -5;   // accepted!
```

```
public class Clock {
    private int second;
    public void setSecond(int s) {
        if (s >= 0 && s < 60) second = s;
    }
}
```

Question	`public` field	`private` field + methods
Who can write which value?	Anyone, any value	Only a value the method accepts
Where is the rule (0–59) checked?	Left to every user; if one forgets, the object is corrupted	In a single place, inside the class
Is a read-only field possible?	No (except with `final`)	Yes; only the getter is written
If the name or type of the field changes?	All the classes that use it change	The signatures of the methods can stay the same

Table 4.2.7: Comparison of the `public` field and the `private` field with methods.

The last row of the table shows the longest-term gain: even when the internal structure changes, the signatures of the methods can be kept. When the two fields of the clock are reduced to a single `totalSeconds` field, the code that uses it does not change at all.

The interactive exercise in the slides makes this comparison on a `DigitalClock` object. The object is in the state 09:05:30. When one wants to write 75 to the seconds, two ways can be followed. The setter way (`s.setSecond(75)`) rejects the value and the clock stays at 09:05:30. Had the field been

`public` (`s.second = 75`), the value would be written directly and the clock would show a meaningless state such as 09:05:75. The valid ranges are 0–23 for the hour and 0–59 for the minute and second.

For classes that only carry data and will never change, since Java 16 there is a shorter way: the **record**. The line `record Point(int x, int y) { }` defines a class whose fields are automatically `private final`. The names of the accessors do not start with `get`; they carry the name of the field: `x()` and `y()`. The compiler also writes the `toString`, `equals` and `hashCode` methods itself:

```
public class RecordTest {
  public static void main(String[] args) {
    Point n = new Point(3, 4);
    System.out.println(n.x() + " " + n.y());
    System.out.println(n);
    System.out.println(n.equals(new Point(3, 4)));
  }
}
```

```
3 4
Point[x=3, y=4]
true
```

An attempt to write to a record's field from outside (`n.x = 5;`) gives the error `x has private access in Point`. Because the field is both `private` and `final`, a record is a ready-made example of an immutable class.

4.2.5 Example: a bank account

The pieces so far can be combined in a single class. A bank account is a good example for this: its data is valuable, its rules are clear and a breach of the rules is easily noticed. The balance must never fall below zero, and money must enter and leave only through certain operations. The `BankAccount` class is examined in four parts.

4.2.5.1 Fields and constructor

```
public class BankAccount {
  private int accountNumber;  // account number
  private double balance;      // account balance
  private String accountType;   // "Checking" / "Savings"

  public BankAccount(int accountNumber,
      double balance, String accountType) {
    this.accountNumber = accountNumber;
    this.balance = balance;
    this.accountType = accountType;
  }

  public int getAccountNumber() {
    return accountNumber;
  }
  public double getBalance() {
    return balance;
  }
  public String getAccountType() {
    return accountType;
  }
  // ... deposit and withdraw methods
```

All three fields of the class are `private`: the account number, the balance and the account type. No one can change the account number or the balance directly. The **constructor** builds the object with its initial values. Because the parameter names are chosen to be the same as the fields, the assignments are written with the `this.balance = balance;` pattern.

The class has only getters and no setter. The `setBalance` method has deliberately not been written. The balance changes only through deposit, withdrawal, interest and transfer operations. In this way every path that changes the balance passes through a method that enforces its own rule. There is no setter for the account number or the account type either; these two fields do not change after the account is opened.

4.2.5.2 Deposit and withdrawal

```
// ...
// deposit
public void deposit(double amount) {
  if (amount > 0) {
    balance += amount;
    System.out.println(amount
        + " TL deposited. New balance: " + balance);
  } else {
    System.out.println("Invalid deposit amount.");
  }
}

// withdraw
public void withdraw(double amount) {
  if (amount <= balance && amount > 0) {
    balance -= amount;
    System.out.println(amount
        + " TL withdrawn. New balance: " + balance);
  } else if (amount > balance) {
    System.out.println("Insufficient balance.");
  } else {
    System.out.println("Invalid withdrawal amount.");
  }
}
```

Each operation first checks the precondition. For a deposit the condition is `amount > 0`. For a withdrawal there are two conditions: `amount > 0` and `amount ≤ balance`. If the condition is broken the balance does not change, only a message is printed. This class has chosen a path between "silently ignoring" and "reporting" when the precondition is broken: it throws no exception, but it prints on the screen what went wrong.

The order of the `if` – `else if` – `else` chain in `withdraw` matters. First the case in which the operation is valid is checked. If it is not valid, the first possible cause is that the amount exceeds the balance. The only case left is that the amount is zero or negative. Thanks to this arrangement every invalid call gets exactly one message.

Together, the two methods protect the class invariant: the balance cannot fall below zero by any route. A deposit only increases the balance; a withdrawal decreases it only if there is enough balance. If the invariant holds when the object is built, it stays true after these two methods as well.

4.2.5.3 Interest

```
  // ...
  // add interest (savings only)
  public void addInterest(double rate) {
    if (accountType.equals("Savings") && rate > 0) {
      double interest = balance * rate / 100;
      balance += interest;
      System.out.println("Adding interest: " + interest
          + " TL. New balance: " + balance);
    } else if (!accountType.equals("Savings")) {
      System.out.println(
          "Interest applies only to savings accounts.");
    } else {
      System.out.println("Invalid interest rate.");
    }
  }
```

Interest is added only to a savings account and with a positive rate. Because the account type is a text, the comparison is made with `equals`, not `==`. `==` checks whether two references point to the same object; `equals` compares the contents of the texts. Two `"Savings"` texts created in two different ways may come out different with `==`.

`interest` is a **local variable**. It exists only while the method runs and is not part of the object. The interest amount is computed once, added to the balance and vanishes when the method ends. Only the three fields make up the object's state.

4.2.5.4 Transfer

```
  // ...
  // transfer to another account
  public void transfer(BankAccount target,
                       double amount) {
    if (amount <= balance && amount > 0) {
      balance -= amount;
      target.deposit(amount);
      System.out.println(amount
          + " TL transferred. New balance: " + balance);
    } else {
      System.out.println(
          "Insufficient balance or invalid amount.");
    }
  }
}
```

The `transfer` method calls a method of another object of the same class: `target.deposit(amount)`. It deducts from its own balance and adds to the target account with the deposit method (Figure 4.2.10).

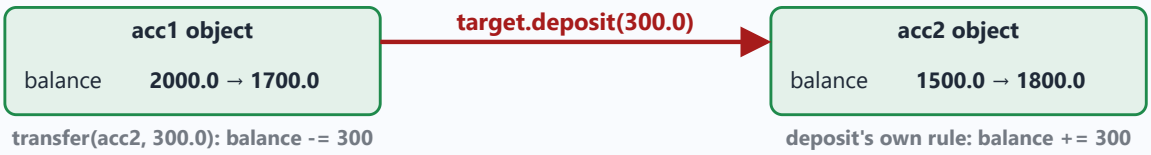


Figure 4.2.10: The two halves of a transfer. `acc1` deducts 300 from its own balance. Then it calls the `deposit` method of the `acc2` object; `acc2` does the adding by its own rule.

It would also have been possible to write directly to the `balance` field of the target account: `target.balance += amount;`. The two objects are of the same class and `private` works at class level. But calling a method keeps the rules in one place. If a rule were added to deposits tomorrow, for example if every deposit were logged, the transfer would automatically pass through this rule too. Code that changed the field directly would skip the new rule.

4.2.5.5 Tracing two accounts step by step

All the parts of the class are run with the `main` method below:

```
// in main:
BankAccount acc1 =
    new BankAccount(123456, 2000.0, "Savings");
BankAccount acc2 =
    new BankAccount(789012, 1500.0, "Checking");
acc1.deposit(500.0);
acc2.withdraw(200.0);
acc1.addInterest(5.0);
acc1.transfer(acc2, 300.0);
System.out.println(acc1.getBalance());
System.out.println(acc2.getBalance());
```

```
500.0 TL deposited. New balance: 2500.0
200.0 TL withdrawn. New balance: 1300.0
Adding interest: 125.0 TL. New balance: 2625.0
300.0 TL deposited. New balance: 1600.0
300.0 TL transferred. New balance: 2325.0
2325.0
1600.0
```

Step	Line	Call stack	Variables	New output
1	2: `BankAccount acc1 =` 3: `new BankAccount(123456, 2000.0, "Savings");`	main()	acc1=→BankAccount	—
2	4: `BankAccount acc2 =` 5: `new BankAccount(789012, 1500.0, "Checking");`	main()	acc1=→BankAccount, acc2=→BankAccount	—
3	6: `acc1.deposit(500.0);`	main() → acc1.deposit(500.0)	amount=500.0	`500.0 TL deposited. New balance: 2500.0`
4	7: `acc2.withdraw(200.0);`	main() → acc2.withdraw(200.0)	amount=200.0	`200.0 TL withdrawn. New balance: 1300.0`
5	8: `acc1.addInterest(5.0);`	main() → acc1.addInterest(5.0)	rate=5.0, interest=125.0	`Adding interest: 125.0 TL. New balance: 2625.0`
6	9: `acc1.transfer(acc2, 300.0);`	main() → acc1.transfer(acc2, 300.0)	amount=300.0, target=→acc2	—
7	9: `acc1.transfer(acc2, 300.0);`	main() → acc1.transfer(acc2, 300.0) → acc2.deposit(300.0)	amount=300.0	`300.0 TL deposited. New balance: 1600.0`
8	9: `acc1.transfer(acc2, 300.0);`	main() → acc1.transfer(acc2, 300.0)	amount=300.0	`300.0 TL transferred. New balance: 2325.0`
9	10: `System.out.println(acc1.getBalance());` 11: `System.out.println(acc2.getBalance());`	main()	acc1=→BankAccount, acc2=→BankAccount	`2325.0 ⏎ 1600.0`

Table 4.2.8: Trace of two bank accounts. The line numbers refer to the `main` fragment above (line 1 is a comment).

In the first two steps two independent objects are built. Each object has its own `balance` field: `acc1`'s is 2000.0 and `acc2`'s is 1500.0. In the third step `acc1.deposit(500.0)` is called. In this call `this` points to `acc1`; `amount > 0` is true and the balance becomes 2500.0. In the fourth step `this` is `acc2` this time. Because 200 ≤ 1500, the withdrawal is made and the balance drops to 1300.0. `acc1` is not affected by this operation.

In the fifth step the interest is computed. The account is a savings account and the rate is positive: interest = 2500.0 × 5 / 100 = 125.0. The balance becomes 2625.0. In the sixth step the transfer begins. The condition is true; `acc1` first deducts from its own balance: 2625.0 − 300.0 = 2325.0. In the seventh step `target.deposit(amount)` is called. There are now three frames on the stack: `main`, `transfer` and `deposit`. In this topmost frame `this` points to `acc2` and its balance becomes 1600.0 (Figure 4.2.11).

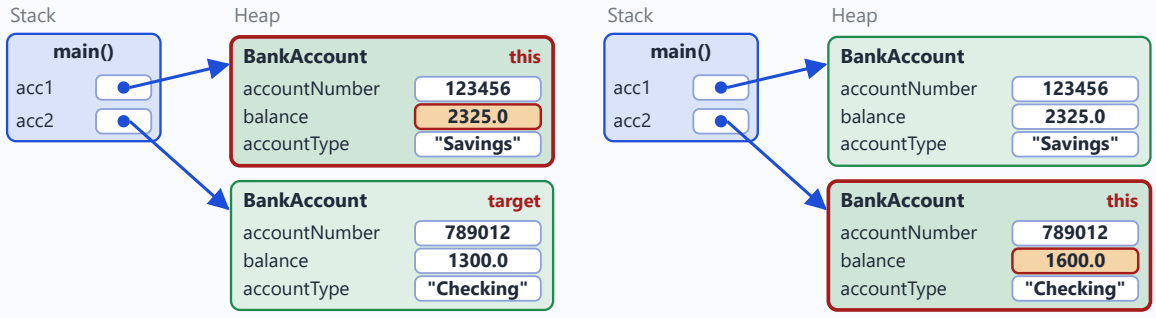


Figure 4.2.11: Two moments inside the transfer. Left: `transfer` is running; `this` points to the first account and `target` to the second; the balance of the first account has dropped to 2325.0. Right: `deposit` is running; this time `this` is the second account and its balance is 1600.0.

In the eighth step `deposit` ends and `transfer` continues where it left off. Note the order in the output: first "300.0 TL deposited" (the message of the inner call), then "300.0 TL transferred" (the message of the outer method) is printed. In the last step the getters read the two balances. The total money has been preserved: 2325 + 1600 = 3925 = 2625 + 1300. A transfer produces no money, it only moves it.

Every method of the class does a constant number of operations; there are no loops. So the time cost of each call is O(1). In memory each account object holds only three fields; as the number of objects grows, memory grows linearly.

The numbers in the output are the default Java form of `double` values: 500.0, 2625.0. In this example all the amounts are whole numbers, so no problem arises. But `double` cannot always hold decimal fractions exactly. When 0.2 TL is deposited into an account with a balance of 0.1 TL using the same class, JDK 21 prints `New balance: 0.30000000000000004`. Real banking software therefore keeps money as an integer number of cents or with the `BigDecimal` class.

4.2.5.6 A constructor that validates too

The operation methods of `BankAccount` protect the balance. But its constructor assigns the incoming values without checking them at all. This gap can be abused in two ways. `new BankAccount(1, -500.0, "Checking")` builds an account with a negative balance; the invariant is already broken as the object is born. If `"savings"` (in lowercase) is written, the account can never receive interest and no one notices. When tried with JDK 21, both objects are built without trouble; when `addInterest(5.0)` is called on the lowercase account, the message "Interest applies only to savings accounts." appears.

The class invariant must hold at the moment the object is born. The constructor is also a precondition checker:

```
public BankAccount(int accountNumber,
        double balance, String accountType) {
    if (balance < 0)
        throw new IllegalArgumentException(
            "Initial balance cannot be negative");
    if (!accountType.equals("Savings")
            && !accountType.equals("Checking"))
        throw new IllegalArgumentException(
            "Unknown account type: " + accountType);
    this.accountNumber = accountNumber;
    this.balance = balance;
    this.accountType = accountType;
}
```

With this constructor, the call `new BankAccount(1, -500.0, "Checking")` throws an exception with the message `Initial balance cannot be negative`, and the call `new BankAccount(2, 1000.0, "savings")` with the message `Unknown account type: savings`. If `null` is given as the account type, the call `accountType.equals(...)` throws a `NullPointerException`; that also prevents the object from being built, but its message is less descriptive.

If the constructor throws an exception, the `new` expression produces no reference. The invalid object never reaches the user. This behavior can be followed with a small `Temperature` class:

```
class Temperature {
    private final double kelvin;
    public Temperature(double kelvin) {
        if (kelvin < 0)
            throw new IllegalArgumentException(
                "Kelvin cannot be negative");
        this.kelvin = kelvin;
    }
    public double getKelvin() { return kelvin; }
}
// in main:
Temperature a = new Temperature(300);
Temperature b = null;
try {
    b = new Temperature(-5);
} catch (IllegalArgumentException e) {
    System.out.println("Rejected: "
        + e.getMessage());
}
System.out.println(a.getKelvin() + " " + b);
```

```
Rejected: Kelvin cannot be negative
300.0 null
```

Step	Line	Call stack	Variables	New output
1	12: `Temperature a = new Temperature(300);`	main()	a=→#1	—
2	13: `Temperature b = null;`	main()	a=→#1, b=null	—
3	15: `b = new Temperature(-5);` 3: `public Temperature(double kelvin) {`	main() → Temperature(-5.0)	a=→#1, b=null, kelvin=-5.0	—
4	4: `if (kelvin < 0)` 5: `throw new IllegalArgumentException(`	main() → Temperature(-5.0)	a=→#1, b=null, kelvin=-5.0	—
5	16: `} catch (IllegalArgumentException e) {` 17: `System.out.println("Rejected: "` 18: `+ e.getMessage());`	main()	a=→#1, b=null	`Rejected: Kelvin cannot be negative`
6	20: `System.out.println(a.getKelvin() + " " + b);`	main()	a=→#1, b=null	`300.0 null`

Table 4.2.9: Trace of validation in the constructor. Line numbers: 1–10 the `Temperature` class, 12–20 the code in `main`.

In the first step 300 is valid; the condition is false, `kelvin` is assigned and `a` points to the new object. In the second step `b` is explicitly initialized with `null`. This assignment is necessary: `b` will be read after the `catch` block. Java does not allow a local variable that might not have a value to be read.

In the third step `new` allocates the second object and calls the constructor. The `kelvin` field of the allocated object is 0.0, the default value; the parameter is −5.0. In the fourth step the precondition is found broken and `throw` cuts the constructor short. The assignment line never runs. Figure 4.2.12 shows these two moments.

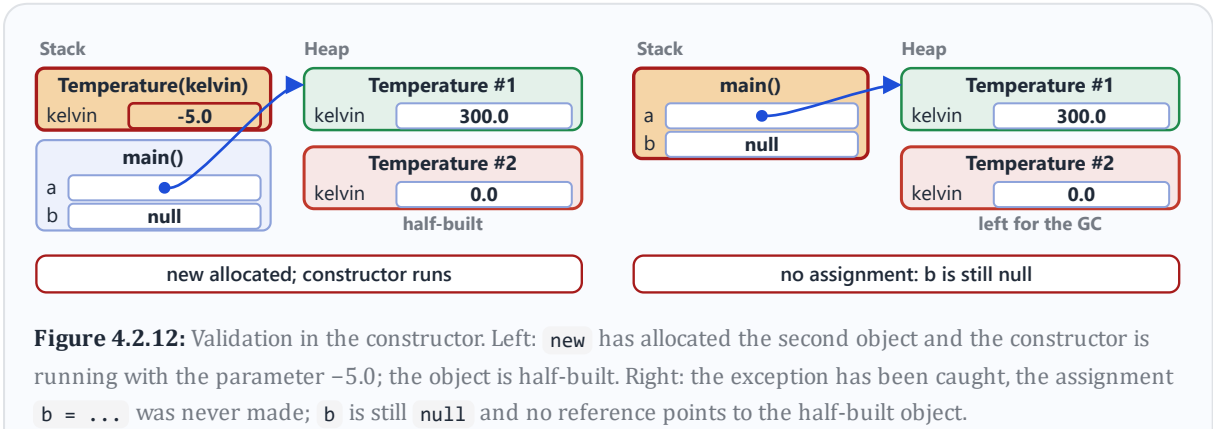


Figure 4.2.12: Validation in the constructor. Left: `new` has allocated the second object and the constructor is running with the parameter −5.0; the object is half-built. Right: the exception has been caught, the assignment `b = ...` was never made; `b` is still `null` and no reference points to the half-built object.

In the fifth step control passes to the `catch` block and `Rejected: Kelvin cannot be negative` is printed. Because the assignment `b = new Temperature(-5);` was cut short, `b` has not changed. No reference reaches the half-built object; the garbage collector cleans it up after a while. In the last step `300.0 null` is printed. Only valid `Temperature` objects can be used in the program: the invariant (`kelvin ≥ 0`) is true for every object.

The constructor is the single door through which an object is built. If the rules are enforced there once, the other methods of the class can trust that the object they receive is valid. In classes that have setters, it is also a common practice for the constructor to call the setter instead of rewriting the same check. That way the rule stays in a single method.

Common mistake

Validating the setters and forgetting the constructor. `setSecond` rejects 75, but if the constructor writes `this.second = second;`, then `new DigitalClock(75)` builds an invalid clock. Every path by which a value enters the object, the constructor included, must pass through the same rule.

4.2.6 Class design

An encapsulated class has two faces: the outer face the user sees and the internal structure only the author knows. This distinction guides most of the decisions made when designing a class. This section defines the distinction as a concept, makes it concrete with Java's `interface` construct, and shows how a class can change its internal structure without breaking its users.

4.2.6.1 Interface and implementation

Definition 4.2.9

A class definition is divided into two parts. The **interface** states what the class does and how it is used: the names, parameters, return types and comments of the `public` methods. The **implementation** contains the internal mechanical details of the class and its `private` fields; these matter only inside the class.

Even if the implementation changes, the interface may stay the same, because private fields are hidden from outside the class. Some sources describe this distinction with three elements. The **specification** is the signatures of the constructor and methods. The **representation** is the `private` fields that hold the data. The **implementation** is the bodies of the constructor and methods. The user knows only the specification. Even if the representation and implementation change, the user's code does not break.

An espresso machine shows this distinction well (Figure 4.2.13). The user knows the "Start", "Water amount" and "Stop" buttons on the control panel. The pump, boiler, valves and brew group are behind the panel. Even if the manufacturer replaces the pump with a quieter model, the panel stays the same and the user notices the difference only from the sound.

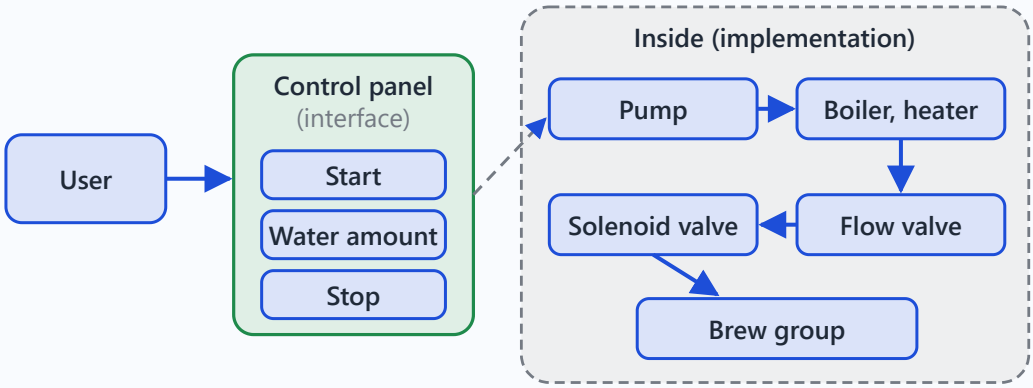


Figure 4.2.13: The espresso machine analogy. The user uses only the control panel (the interface); the pump, boiler and heater, the flow and solenoid valves and the brew group are the internal mechanism (the implementation).

4.2.6.2 interface and implements

Java has set aside a language construct for the idea of an interface. A type declared with the `interface` keyword is a contract that says what a class will do. The methods have only a signature, no body:

```
public interface Car {
    // Interface: WHAT is done?
    public void drive();

    public void stop();
}
```

The methods in an interface are implicitly `public` and `abstract`; the meaning does not change if the `public` words in the example are left out. The person driving the car knows only the methods `drive()` and `stop()`. The class that implements the interface writes how the job is done:

```
public class SportsCar implements Car {
  // Implementation: HOW is it done?
  private String brand;
  private int speed;

  public SportsCar(String brand) {
    this.brand = brand;
  }

  public void drive() {
    speed = 100;
    System.out.println(brand + " is moving: "
        + speed + " km/sa");
  }

  public void stop() {
    speed = 0;
    System.out.println(brand + " stopped.");
  }
}
```

```
// in main:
Car car = new SportsCar("Rüzgar");
car.drive();
car.stop();
```

```
Rüzgar is moving: 100 km/h
Rüzgar stopped.
```

The phrase `implements Car` says that `SportsCar` implements the `Car` interface. This is a promise, and the compiler checks whether the promise is kept: the class must write every method in the interface. Figure 4.2.14 shows the relation between the two types in UML.

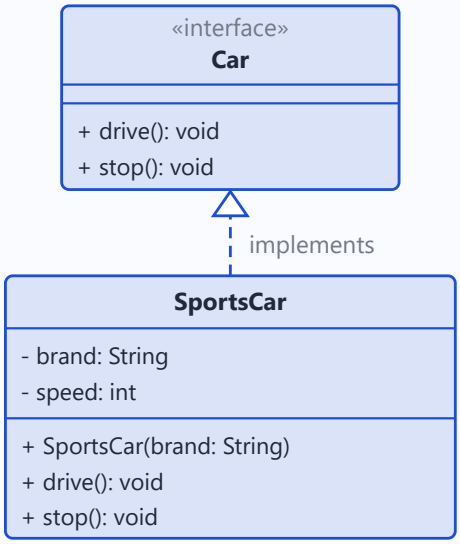


Figure 4.2.14: In UML notation the `SportsCar` class implements the `Car` interface. The dashed line with an empty triangle is the "implements" relation. The interface has only two `public` methods; the class additionally has two `private` fields and a constructor.

An interface and a class cannot have the same name in the same package. If one tries to give the class the name `Car` too, the compiler gives the error `duplicate class: Car`; that is why the class is named `SportsCar`. The `main` method declares its variable with type `Car`: `Car car = new SportsCar("Rüzgar");`. This code therefore depends only on the interface. If another implementation were used in place of `SportsCar` tomorrow, only the `new` expression would change.

What happens if a method is not written? If `SportsCar` writes only `drive()` and forgets `stop()`, JDK 21 gives this message:

```
MissingMethod.java:5: error: SportsCar is not abstract and does not override abstract
method stop() in Car
class SportsCar implements Car {
^
1 error
```

The message also names both ways out: either the class writes the missing method or it is itself marked `abstract`. What matters here is that the interface turns the "outer face" idea of encapsulation into a construct the compiler checks.

4.2.6.3 Changing the implementation

The strongest gain of encapsulation is being able to change the internal structure of a class without breaking its users. `DigitalClock` is written in two versions to show this. The old version keeps the minute and the second in two separate fields:

```
public class DigitalClock {
  private int minute = 0;
  private int second = 0;
  public int getMinute() {
    return minute;
  }
  public int getSecond() {
    return second;
  }
  public void setMinute(int minute) {
    if (minute >= 0 && minute < 60) {
      this.minute = minute;
    }
  }
  public void setSecond(int second) {
    if (second >= 0 && second < 60) {
      this.second = second;
    }
  }
}
```

The new version uses a single field: the number of seconds since midnight. The minute and second are no longer stored; they are computed from this number every time they are asked for:

```
public class DigitalClock {
  // seconds since midnight
  private int totalSeconds = 0;
  public int getMinute() {
    return totalSeconds / 60 % 60;
  }
  public int getSecond() {
    return totalSeconds % 60;
  }
  public void setMinute(int minute) {
    if (minute >= 0 && minute < 60) {
      int diff = minute - getMinute();
      totalSeconds += diff * 60;
    }
  }
  public void setSecond(int second) {
    if (second >= 0 && second < 60) {
      totalSeconds += second - getSecond();
    }
  }
}
```

The `public` methods of the two versions are the same: `getMinute()`, `getSecond()`, `setMinute(int)`, `setSecond(int)`. The code that uses them cannot tell which one it is working with. The computations in the new version need a little thought. `totalSeconds / 60` gives the total number of minutes; `% 60` separates its minute part within the hour. `totalSeconds % 60` is the second part. The setters cannot write the new value directly, because they must preserve the other part. Instead they add the difference between the old value and the new value.

The usage code below is traced step by step with the new version:

```
private int totalSeconds = 0;  // one field
public int getSecond() {
    return totalSeconds % 60;
}
public void setSecond(int second) {
    if (second >= 0 && second < 60)
        totalSeconds += second - getSecond();
}
// getMinute(): totalSeconds / 60 % 60
DigitalClock s = new DigitalClock(); // main
s.setMinute(12);
s.setSecond(45);
s.setSecond(75);
s.setSecond(9);
System.out.println(s.getMinute() + ":"
    + s.getSecond());
```

Step	Line	Variables	New output
1	10: `DigitalClock s = new DigitalClock(); // main`	totalSeconds=0	—
2	11: `s.setMinute(12);`	totalSeconds=720	—
3	12: `s.setSecond(45);` 5: `public void setSecond(int second) {` 6: `if (second >= 0 && second < 60)`	second=45, totalSeconds=720	—
4	7: `totalSeconds += second - getSecond();` 2: `public int getSecond() {` 3: `return totalSeconds % 60;`	second=45, getSecond()=0, totalSeconds=765	—
5	13: `s.setSecond(75);` 6: `if (second >= 0 && second < 60)`	second=75, totalSeconds=765	—
6	14: `s.setSecond(9);` 7: `totalSeconds += second - getSecond();`	second=9, getSecond()=45, totalSeconds=729	—
7	15: `System.out.println(s.getMinute() + ":"` 16: `+ s.getSecond());`	getMinute()=12, getSecond()=9	`12:9`

Table 4.2.10: Trace of the single-field `DigitalClock`. Line numbers refer to the shortened code above: 1–9 are fragments of the class, 10–16 are `main`.

At the start `totalSeconds` is 0; to an outside observer the clock reads 00:00. In the call `setMinute(12)`, `getMinute()` returns 0, the difference is 12 minutes and 12 · 60 = 720 seconds are added. The old version would have written just `minute = 12` in the same call. In the call `setSecond(45)`, first `getSecond()` = 720 % 60 = 0 is computed; the difference is 45 and the total becomes 765. To an outside observer the clock reads 12:45.

The call `setSecond(75)` is rejected in both versions; the rule (0–59) has stayed inside the class. In the call `setSecond(9)` the difference is negative: 9 − 45 = −36. The total becomes 765 − 36 = 729 and the minute part is still 12. Figure 4.2.15 shows these two moments together with the two versions.

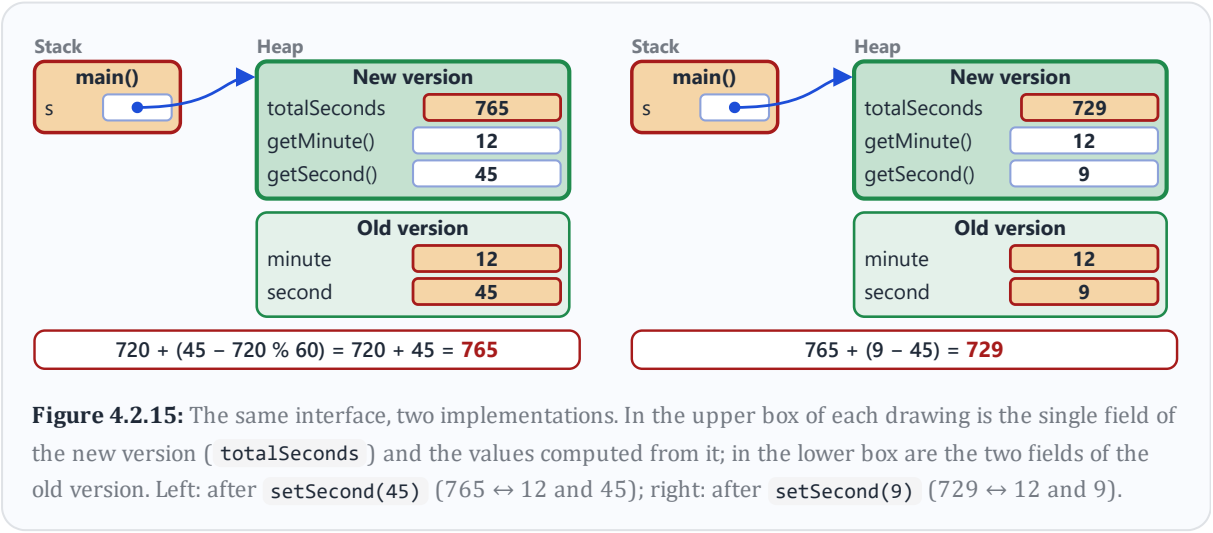


Figure 4.2.15: The same interface, two implementations. In the upper box of each drawing is the single field of the new version (`totalSeconds`) and the values computed from it; in the lower box are the two fields of the old version. Left: after `setSecond(45)` (765 ↔ 12 and 45); right: after `setSecond(9)` (729 ↔ 12 and 9).

The last line prints `12:9`; the old version would have given the same. When the two versions are run side by side with the same calls on JDK 21, the same minute and second values are found at every step: 12:45 and 12:9. The internal representation changed, and not a single letter of `main` changed. The notation `12:9` in the output is because no formatting is applied; with `printf("%d:%02d", ...)` one gets `12:09`.

This change has a price too. In the old version `getMinute()` reads just a field. In the new version it does a division and a remainder operation. Both are O(1); the difference is negligible. But the setters now call the getter first. The designer can make such trade-offs without reflecting them to the user at all. This is the freedom encapsulation provides.

4.2.6.4 Design advice

Four pieces of advice are followed when writing an encapsulated class:

1. All the fields in the class are declared `private`.
2. *As many* `public` methods as necessary are provided to access the fields. It is not obligatory to write a setter automatically for every field.
3. Comments are written saying how each method is to be used: what it does and what its preconditions and postconditions are.

4. Helper methods used only inside the class are declared `private`.

The fourth piece of advice is a reminder that encapsulation applies to methods as well as to fields:

```
public class Student {
    private int grade;
    private boolean isValid(int value) {
        return value >= 0 && value <= 100;
    }
    public void setGrade(int grade) {
        if (isValid(grade)) this.grade = grade;
    }
}
```

`isValid` is an internal matter of the class. `setGrade` uses it, but there is no reason to expose it. When `o.isValid(50)` is called from another class, the compiler gives this error:

```
Outside.java:4: error: isValid(int) has private access in Student
        boolean b = o.isValid(50);
                    ^
1 error
```

Keeping the helper method `private` keeps the outer face of the class small. Every method on the outer face is a promise and cannot be taken back later. A `private` method, on the other hand, is a detail that the author can change or delete whenever they like.

4.2.6.5 Tell, don't ask

Writing a getter and a setter for every field is not encapsulation. If the rules are still enforced outside the class, the data has merely been moved one step away. The two code fragments below show the difference. In the first, the code that uses the object **asks** the object for its data and makes the decision itself:

```
// same check at every withdrawal
if (h.getBalance() >= 300) {
    h.setBalance(h.getBalance() - 300);
}
// ...and wherever points are added
o.setGrade(Math.min(100, o.getGrade() + 5));
```

In this form the rule is repeated in every place that uses the object. If one place forgets it, the object is corrupted. Moreover, as soon as `setBalance` is written, anyone can set the balance to any value they want; the class's invariant is left unprotected. In the second form the code that uses the object **tells** it what is wanted:

```
h.withdraw(300);   // rule in BankAccount
o.addBonus(5);     // 100 cap in Student
```

The decision is made by the object that holds the data. `BankAccount` decides whether a 300 TL withdrawal will be made; `Student` ensures that the grade does not exceed 100. A setter is written only where the field can truly change freely. That is why `BankAccount` has no `setBalance`.

A simple question is enough to find out where a method belongs: "Who has the data needed to make this decision?" If the answer is the object itself, the decision should also be in its method.

4.2.6.6 The gains of encapsulation

The examples in these notes up to here have shown the four gains of encapsulation concretely:

- **Abstraction.** The relation between the object and its user becomes simpler; the user knows only the methods. `service.addData("Ayşe")` is a single line.
- **Protection.** The object is protected against unwanted or faulty access by users. `s.second = 75;` does not compile, and `setSecond(75)` is rejected.
- **Changeability.** The internal structure of the class can be changed without changing the interface; the code that uses it is not affected. The two-field clock turned into a single-field clock and `main` stayed the same.
- **Valid state.** The state of the object always obeys the rules. Only points with positive coordinates, seconds and minutes between 0 and 59, non-negative hours or balances. `withdraw` never lets the balance fall below zero.

The large-scale result of these gains is seen in Figure 4.2.16. A program may have many classes that use `BankAccount`: banking operations, reports, tests. All of them depend only on the interface.

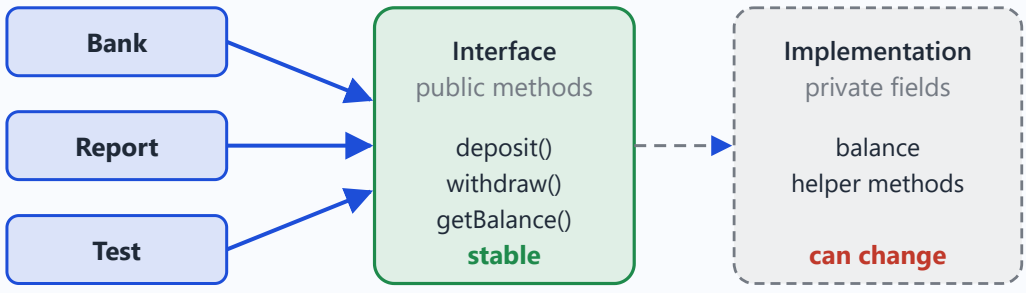


Figure 4.2.16: The direction of dependency. The `Bank`, `Report` and `Test` classes depend only on the interface (`deposit()`, `withdraw()`, `getBalance()`); the interface is stable. The implementation (private fields and helper methods) is behind the interface and may change.

Interfaces change much more rarely than implementations; this keeps the code stable. When an implementation changes, the code that depends on it directly has to change too. Hiding the implementation removes this dependency and the change stays inside the class. This situation is called **loose coupling**. In a loosely coupled program, changing one class does not require touching the others.

4.2.7 Immutable classes

Making fields `private` and writing no setters does not mean encapsulation is complete. If a field points to a mutable object such as an array or a list, the reference to that object can leak out. This section explains how the leak arises, how it is closed, and why objects that never change solve this problem from the start.

4.2.7.1 A leaked reference

The `ReportCard` class keeps the grades in an array. The field is `private final` and there is no setter. At first sight the grades look protected:

```
public class ReportCard {
    private final int[] grades;
    public ReportCard(int[] grades) {
        this.grades = grades; // same array
    }
    public int[] getGrades() {
        return grades;        // inner array
    }
    public double average() {
        int t = 0;
        for (int n : grades) t += n;
        return (double) t / grades.length;
    }
}
```

```
int[] arr = {70, 80, 90};
ReportCard k = new ReportCard(arr);
arr[0] = 0;                // outside array
k.getGrades()[1] = -100;    // from the getter
System.out.println(k.average());
```

```
-3.3333333333333335
```

The grades have been changed from outside; moreover an invalid grade like −100 has got in. The reason is that an array is an object. The constructor and the getter share not the array but only its reference. The same array can be reached from three places: from the variable `arr` in `main`, from the field of `ReportCard` and from the reference the getter returns. Table 4.2.11 shows step by step how this situation arises.

Step	Line	Call stack	Variables	New output
1	8: `int[] arr = {70, 80, 90};`	main()	arr=→[70, 80, 90]	—
2	9: `ReportCard k = new ReportCard(arr);` 2: `this.grades = grades;`	main() → ReportCard(grades)	arr=→[70, 80, 90], k=→ReportCard	—
3	10: `arr[0] = 0;`	main()	arr=→[0, 80, 90], k=→ReportCard	—
4	11: `k.getGrades()[1] = -100;` 5: `return grades;`	main()	arr=→[0, -100, 90], k=→ReportCard	—
5	12: `System.out.println(k.average());`	main()	arr=→[0, -100, 90], k=→ReportCard	`-3.3333333333333335`

Table 4.2.11: Trace of the leaked reference. Line numbers: 1–6 the constructor and getter of `ReportCard`, 8–12 `main`.

In the first step the array is created on the heap; the variable `arr` holds only its reference. In the second step the constructor writes the reference in the parameter to the field. Now the `private` field of `ReportCard` and the variable `arr` of `main` point to the same array. The constructor has copied the reference, not the array.

In the third step `arr[0] = 0` runs. This line does not pass through any method of `ReportCard`, yet the grade that `ReportCard` sees changes. In the fourth step `getGrades()` returns the inner array itself and −100 is written through the returned reference. An invalid grade has got into the object without passing any check (Figure 4.2.17).

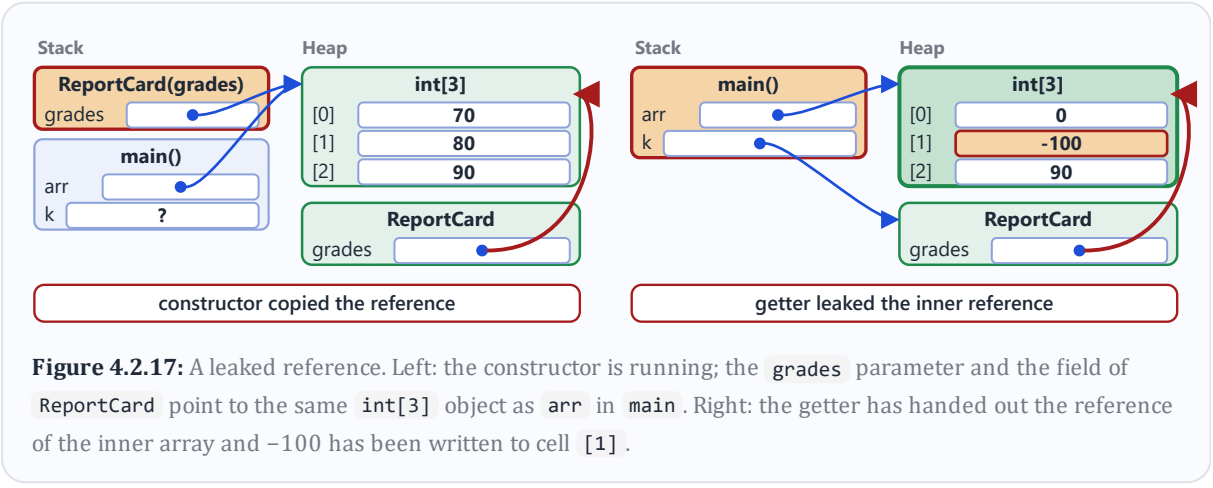


Figure 4.2.17: A leaked reference. Left: the constructor is running; the `grades` parameter and the field of `ReportCard` point to the same `int[3]` object as `arr` in `main`. Right: the getter has handed out the reference of the inner array and −100 has been written to cell `[1]`.

In the last step the average (0 − 100 + 90) / 3 ≈ −3.33 is found. Java prints this value as `-3.3333333333333335`; the 5 in the last digit comes from the binary representation of `double`. Writing `private` was not enough: the reference to the inner object leaked out.

The word `final` does not prevent this either. `final` only prevents the reference from pointing to another array; it does not protect the contents of the array. Inside the class `grades[0] = 0;` compiles without trouble, whereas `grades = new int[2];` gives the error `cannot assign a value to final variable grades`. What does not change is the reference in the box, not the object the reference points to.

4.2.7.2 Defensive copying

Definition 4.2.10

A **defensive copy** is copying a mutable object as it enters the class and as it leaves the class. The only reference that reaches the inner object remains the class's own field.

```
public class ReportCard {
    private final int[] grades;
    public ReportCard(int[] grades) {
        this.grades = grades.clone(); // on entry
    }
    public int[] getGrades() {
        return grades.clone();        // on exit
    }
    // average() unchanged
}
```

With the same `main` method this version prints:

```
80.0
```

The `grades.clone()` in the constructor protects against the caller changing the array it holds afterwards. `arr[0] = 0` now changes only the array of `main`; the copy of `ReportCard` stays at 70. The copy in the getter protects against the returned array being changed. `k.getGrades()[1] = -100` writes to a temporary copy, and that copy is lost immediately. The average is (70 + 80 + 90) / 3 = 80.0.

For lists there are two options. `new ArrayList<>(list)` produces an independent copy. `Collections.unmodifiableList(list)` does not copy; it gives a read-only **view** of the same list. The difference between the two can be seen with JDK 21 as follows:

```
import java.util.*;
public class Lists {
  public static void main(String[] args) {
    List<Integer> inner = new ArrayList<>(List.of(70, 80));
    List<Integer> view = Collections.unmodifiableList(inner);
    List<Integer> copy = new ArrayList<>(inner);
    inner.add(90);
    System.out.println(view + " " + copy);
    try {
      view.add(100);
    } catch (UnsupportedOperationException e) {
      System.out.println(e.getClass().getName());
    }
    List<Integer> fixed = List.copyOf(inner);
    inner.set(0, 0);
    System.out.println(fixed + " " + view);
  }
}
```

```
[70, 80, 90] [70, 80]
java.lang.UnsupportedOperationException
[70, 80, 90] [0, 80, 90]
```

After 90 is added to the inner list, the view shows `[70, 80, 90]` and the copy shows `[70, 80]`. The view reflects the changes of the inner list; only changes through the view cannot be made. `view.add(100)` throws an `UnsupportedOperationException`. `List.copyOf`, available since Java 10, produces an unmodifiable copy: when the first element of the inner list is made 0, the copy is not affected.

Immutable objects such as `String` and `Integer` are not copied. Since there is no way to change them, sharing them is harmless. Copying also has a cost: a copy of an array of n elements needs O(n) time and O(n) extra memory. Because a copy is produced every time the getter is called, this cost accumulates for large arrays that are called often. In such cases a read-only view or an immutable type from the start is cheaper.

4.2.7.3 Designing an immutable class

Definition 4.2.11

An **immutable** object is one whose state never changes after it has been created. `String` is the best-known example.

The way to solve the leaked reference problem at the root is for the object never to change. The `Money` class below keeps the amount as a `long` in cents:

```
public final class Money {        // no subclasses
    private final long cents;     // assigned once
    private final String unit;
    public Money(long cents, String unit) {
        this.cents = cents;
        this.unit = unit;
    }
    public Money plus(Money other) {   // instead of a setter
        if (!unit.equals(other.unit))
            throw new IllegalArgumentException("Units differ");
        return new Money(cents + other.cents, unit);
    }
    @Override public String toString() {
        return cents / 100 + "."
            + String.format("%02d", cents % 100) + " " + unit;
    }
}
```

An immutable class obeys five rules:

1. There are no setters; the state is established only in the constructor.
2. The fields are `private final`.
3. The class is `final`, so a mutable subclass cannot be written.
4. Methods that "change" do not modify the object but return a new one.
5. If there are mutable fields, a defensive copy is made.

The compiler supports every rule as well. When an attempt is made to assign a value to a `final` field outside the constructor, for example when `cents += k;` is written, the error `cannot assign a value to final variable cents` appears. An attempt to derive from a `final` class gives the error `cannot inherit from final Money`. The third rule may seem unnecessary. But if a subclass could be written, that subclass could add its own mutable fields and be used in place of `Money`.

The gains of an immutable class are large. The invariant is checked only in the constructor; afterwards it can never be broken. The object can be shared without copying; no defensive copy is needed. It can safely be used as a key in structures such as `HashMap`, because the key cannot change and lose its place in the table. It is also safe between threads; two threads reading the same object cannot disturb each other. `String` and `record` are ready-made examples of these rules.

The `toString` method writes the amount in lira and kuruş: `cents / 100` gives the whole-lira part and `cents % 100` the kuruş part. `String.format("%02d", ...)` pads the kuruş to two digits; 5 kuruş appears as `0.05 TL`. This notation breaks for negative amounts: `new Money(-50, "TL")` prints `0.-50 TL`, because −50 / 100 = 0 and −50 % 100 = −50. If the class is to allow negative amounts, `toString` has to handle this case separately.

4.2.7.4 Tracing the Money class

```
public Money plus(Money other) {
    if (!unit.equals(other.unit))
        throw new IllegalArgumentException(
            "Units differ");
    return new Money(cents + other.cents, unit);
}
// in main:
Money a = new Money(1250, "TL");
a.plus(new Money(100, "TL"));  // result lost
Money b = a.plus(new Money(375, "TL"));
System.out.println(a + " | " + b);
```

Step	Line	Call stack	Variables	New output
1	8: `Money a = new Money(1250, "TL");`	main()	a=12.50 TL	—

2	9: `a.plus(new Money(100, "TL")); // result lost`
	5: `return new Money(cents + other.cents, unit);`
		main() → a.plus(other)	a=12.50 TL	—

3	9: `a.plus(new Money(100, "TL")); // result lost`	main()	a=12.50 TL	—
4	10: `Money b = a.plus(new Money(375, "TL"));` 5: `return new Money(cents + other.cents, unit);`	main()	a=12.50 TL, b=16.25 TL	—
5	11: `System.out.println(a + " | " + b);`	main()	a=12.50 TL, b=16.25 TL	`12.50 TL | 16.25 TL`

Table 4.2.12: Trace of immutable `Money` objects. Line numbers: 1–6 the `plus` method, 8–11 `main`.

In the first step the first object is built: 1250 kuruş, that is, 12.50 TL. Because its fields are `final`, no method can change them from now on. In the second step `a.plus(new Money(100, "TL"))` is called. A second object (#2) is built as the argument. `plus` does not change `a`; it builds a new object (#3) carrying the sum and returns it.

The third step shows the most frequent mistake. The returned value has not been assigned to a variable; #3 is lost and `a` has not changed (Figure 4.2.18). The compiler gives no warning for this line; ignoring the result of a method that returns a value is allowed in Java. In the fourth step the result is assigned to `b`: 1250 + 375 = 1625 kuruş. The #4 built as the argument is no longer used either. In the last step `a` has kept its first value.

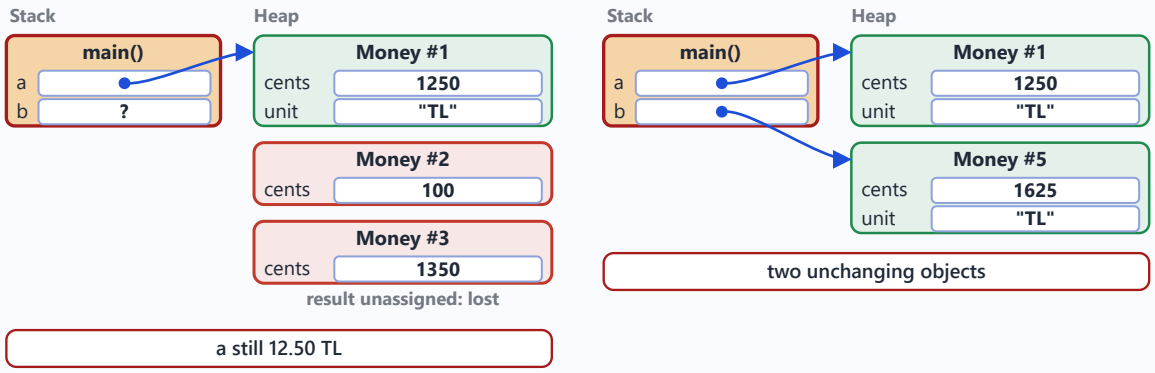


Figure 4.2.18: Immutable objects. Left: the result of `plus` has been discarded: no variable is attached to object #3 (1350 kuruş), and `a` is still 12.50 TL. Right: the end of the program; `a` points to 1250 kuruş and `b` to 1625 kuruş; two separate, unchanging objects.

```
12.50 TL | 16.25 TL
```

The methods of `String` work the same way. The trial below also shows the other behaviors of the `Money` class:

```
public class MoneyDemo {
  public static void main(String[] args) {
    String s = "java";
    s.toUpperCase();
    System.out.println(s);
    s = s.toUpperCase();
    System.out.println(s);
    Money a = new Money(1250, "TL");
    try {
      a.plus(new Money(100, "USD"));
    } catch (IllegalArgumentException e) {
      System.out.println(e.getMessage());
    }
    System.out.println(new Money(5, "TL") + " " + new Money(-50, "TL"));
  }
}
```

```
java
JAVA
Units differ
0.05 TL 0.-50 TL
```

`s.toUpperCase();` on its own does not change `s`, and `java` is printed; when `s = s.toUpperCase();` is written, `JAVA` is obtained. Adding different units is rejected with the message `Units differ`. The last line shows that `toString` prints 5 kuruş correctly and −50 kuruş wrongly. The correct pattern is always the same: `a = a.plus(p);`.

Common mistake

Taking a method of an immutable object for a mutator. Lines such as `a.plus(p);`, `s.toUpperCase();` and `s.trim();` change nothing on their own. The method returns a new object, and if the result is not assigned, it is lost. The compiler does not warn; the error is seen only in the output.

4.2.8 Packages

Encapsulation does not stop at the boundary of a single class. A large program consists of hundreds of classes, and some of these classes serve only a few neighboring classes. Packages group classes, prevent name clashes and build a second layer of encapsulation one level above the class.

4.2.8.1 Packages and folders

Definition 4.2.12

A **package** is a collection of classes grouped in a folder; the name of the folder is the name of the package. Which package a file belongs to is stated by the `package package_name;` declaration on the first line of the file.

Several rules apply when working with packages. Every `public` class is defined in a separate file, in a file with the same name as the class. The `package` declaration is the first line of the file; only comments may come before it. In nested folders a dot is used instead of `/` or `\`: the folder `school/admin` is the package `school.admin`. Package names are written in lowercase. Java's own packages are named this way too: `java.lang`, `java.util`, `java.io`.

The project structure below has two packages. The `bank` package contains the account work and the `app` package the entry point of the program. The source files are under `src/` and the compiled files under `out/`:

```
project/
├── src/
│   ├── bank/
│   │   ├── Account.java      // package bank;
│   │   └── Helper.java   // package bank;
│   └── app/
│       └── Main.java        // package app;
└── out/                    // compiler output
    ├── bank/
    │   ├── Account.class
    │   └── Helper.class
    └── app/
        └── Main.class
```

The **full name** of a class includes its package: `bank.Account`, `app.Main`. The compiler and the Java virtual machine look for packages starting from the roots of the **classpath**. The package name gives the folder path from this root: the class `bank.Account` is the file `bank/Account.class` under the root. The default classpath is the folder in which the command is run (`.`).

Common mistake

Writing a `public` class in a file with a different name. When `public class Account` is written in the file `Bank.java`, the compiler gives the error `class Account is public, should be declared in a file named Account.java`. The file name must be exactly the same as the name of the `public` class, including case.

4.2.8.2 Compiling and running a packaged program

Let us write a small program with two packages. The `bank.Account` class does deposits and balance reading; `app.Main` uses this class:

```
package bank;

public class Account {
    private double balance;

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public double getBalance() {
        return balance;
    }
}
```

```
package app;

import bank.Account;

public class Main {
    public static void main(String[] args) {
        Account h = new Account();
        h.deposit(250.0);
        System.out.println("Balance: " + h.getBalance());
    }
}
```

To be able to use `Account` from another package by its short name, `Main` writes the line `import bank.Account;`. In the project folder the compile and run commands are:

```
> javac -d out src/bank/Account.java src/app/Main.java
> java -cp out app.Main
Balance: 250.0
```

The `-d out` option writes the `.class` files under `out/` together with the package folders. When run with JDK 21, the files `out/bank/Account.class` and `out/app/Main.class` are produced. The `Helper.class` shown in the project structure appears only if `Helper.java` is also added to the compile command. `-cp out` sets the classpath to the `out` folder. When running, not the file path of the class but its **full name** is given: `app.Main`.

This distinction is the point where beginners get stuck most often. When wrong commands are tried in the same project, JDK 21 gives these messages:

Command	Result	Why
`java -cp out app.Main`	`Balance: 250.0`	Classpath and full name are correct.
`java -cp out Main`	`ClassNotFoundException: Main`	The package name is missing; there is no `Main.class` at the root.
`java -cp out/app Main`	`NoClassDefFoundError: Main (wrong name: app/Main)`	The file was found, but the class inside it is named `app.Main`.
`java -cp out/app Main.class`	`ClassNotFoundException: Main.class`	The file name was given instead of the class name.

Table 4.2.13: Attempts to run the packaged class `app.Main` (in the project folder, JDK 21).

The message on the third row is instructive. Java finds the file `Main.class`, but the class inside the file introduces itself as `app.Main`. Because the package name does not match the folder structure, the class is not loaded. A package is not only a folder; it is a part of the class's name.

4.2.8.3 Importing (import)

Definition 4.2.13

An **import** is a declaration that makes it possible to use a class from another package by its short name. `import package_name.ClassName;` imports a single class, and `import package_name.*;` imports all the classes in the package.

`import` is not mandatory; a class can always be written with its full name: `java.util.Random`. `import` only shortens the long name. Some classes need no `import` at all. Classes in the `java.lang` package (`String`, `Math`, `System`, `Integer`) and classes in the same package are visible automatically in every file.

In a source file the order is fixed: first `package`, then the `import`s, and last the class. The `import static` form lets a `static` member of a class be used without writing the class name. The program below shows the three ways together:

```
import java.util.Scanner;
import static java.lang.Math.sqrt;

public class ImportDemo {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    java.util.Random rnd =
        new java.util.Random(42);
    System.out.println(sqrt(n) + " "
        + Math.max(n, 10) + " "
        + rnd.nextInt(100));
  }
}
```

When 49 is given as input, the output is:

```
7.0 49 30
```

`Scanner` is used by its short name thanks to `import java.util.Scanner;`. `java.util.Random`, on the other hand, is written with its full name without being imported. `sqrt` is called without writing `Math.` thanks to `import static java.lang.Math.sqrt;`; the square root of 49 is `7.0`. `Math.max` needs no `import` because it comes from `java.lang`, and it gives the larger of 49 and 10. The last number is the first `nextInt(100)` value of a random number generator started with seed 42. Because the same seed produces the same sequence on every run, this value is always 30.

When `import` is forgotten, the compiler does not recognize the class. In a file that uses `Scanner` without importing it, JDK 21 gives a separate error for each of the two occurrences of `Scanner` in the line:

```
ScannerMissing.java:3: error: cannot find symbol
        Scanner sc = new Scanner(System.in);
        ^
  symbol:   class Scanner
  location: class ScannerMissing
ScannerMissing.java:3: error: cannot find symbol
        Scanner sc = new Scanner(System.in);
                         ^
  symbol:   class Scanner
  location: class ScannerMissing
2 errors
```

The fix is to add `import java.util.Scanner;` at the top of the file. `import` is only a compile-time convenience; it does not slow the program down and puts no extra code in the `.class` file. Writing `import java.util.*;` also does not "load" all the classes in the package into the program; it only

tells the compiler where to look for names.

4.2.8.4 Package access

Classes and members written without an access modifier are **package-private**: they are visible only from their own package. This rule applies to classes as well. The `Helper` class in the `bank` package is not `public`:

```
package bank;

class Helper {     // not public
    static double vat(double amount) {
        return amount * 0.20;
    }
}
```

```
package app;

import bank.Helper;

public class Main2 {
}
```

```
app/Main2.java:3: error: Helper is not public in bank; cannot be accessed from outside
package
import bank.Helper;
            ^
1 error
```

`Helper` is an internal matter of the `bank` package. `Account` in the same package can use it; the `app` package cannot. A package thus also becomes a level of encapsulation. Classes are closed off from the outside; only the `public` ones make up the outer face of the package. Just as a class hides its inside with `private` fields, a package hides its inside with non-`public` classes.

To make a class usable from outside the package, making the class `public` may not be enough. The constructors and methods used must also be `public`. If the `school.Student` class has been made `public` but the `getName()` method has been left without a modifier, the call `o.getName()` from the `app` package gives the error `getName() is not public in Student; cannot be accessed from outside package`.

4.2.8.5 Package naming and subpackages

Millions of Java classes have been written in the world, and a simple package name such as `bank` can easily clash with someone else's. To prevent clashes, organizations begin the package name with their reversed domain name. Because the domain name of Giresun University is `giresun.edu.tr`, the package name becomes `tr.edu.giresun.bank`. Dotted names correspond to folders: `tr/edu/giresun/bank/`.

Names looking nested grants no access right. **A subpackage is a separate package.** The package `tr.edu.giresun.bank.report` cannot see the package-private members of the package `tr.edu.giresun.bank`:

```
// tr/edu/giresun/bank/Helper.java
package tr.edu.giresun.bank;
class Helper {                  // package-private
    static double vat(double t) { return t * 0.20; }
}
```

```
// tr/edu/giresun/bank/report/Report.java
package tr.edu.giresun.bank.report;
import tr.edu.giresun.bank.Helper;
public class Report {
    double compute() { return Helper.vat(100); }
}
```

```
tr/edu/giresun/bank/report/Report.java:3: error: Helper is not public in
tr.edu.giresun.bank; cannot be accessed from outside package
import tr.edu.giresun.bank.Helper;
                          ^
tr/edu/giresun/bank/report/Report.java:5: error: Helper is not public in
tr.edu.giresun.bank; cannot be accessed from outside package
    double compute() { return Helper.vat(100); }
                              ^
2 errors
```

The same rule applies to `import`. `import tr.edu.giresun.bank.*;` does not import subpackages. `import java.util.*;` likewise does not bring in the classes of the `java.util.function` package. When the `Function` interface is to be used, the compiler cannot find the class:

```
FunctionMissing.java:5: error: cannot find symbol
        Function<Integer, Integer> f = x -> x * 2;
        ^
  symbol:   class Function
  location: class FunctionMissing
1 error
```

The fix is to write the line `import java.util.function.Function;` separately. If the package name and the folder path do not match exactly, the compiler and the virtual machine cannot find the class. Development environments therefore create the folders from the package name themselves.

4.2.8.6 Class name clashes

Two classes in different packages can have the same name. A well-known example in the Java library is `java.util.Date` and `java.sql.Date`. If both packages are imported with `*`, the name `Date` becomes ambiguous:

```
import java.util.*;
import java.sql.*;

public class Ambiguous {
    public static void main(String[] args) {
        Date date = null;
    }
}
```

```
Ambiguous.java:6: error: reference to Date is ambiguous
        Date date = null;
        ^
  both class java.sql.Date in java.sql and class java.util.Date in java.util match
1 error
```

The compiler finds both classes and cannot choose which one is meant. To resolve the ambiguity the package name is used. One is imported with a single-class `import` and the other is written with its full name:

```
import java.util.Date;

public class DateDemo {
  public static void main(String[] args) {
    Date start = new Date(0);  // java.util
    java.sql.Date sqlDate =          // full name
        java.sql.Date.valueOf("2024-09-16");
    System.out.println(start.getTime());
    System.out.println(sqlDate);
  }
}
```

```
0
2024-09-16
```

A single-class `import java.util.Date;` takes precedence over imports made with `*`. `Date` now means only `java.util.Date`. `new Date(0)` builds the object denoting midnight on 1 January 1970 (UTC); because `getTime()` gives the milliseconds elapsed since that moment, 0 is printed. `java.sql.Date` is used with its full name; its `valueOf` method produces a date from a text in the form `yyyy-mm-dd` and it is printed in the same form.

4.2.9 Sample problems

The problems in this section are the sample problems and exercises of the slides. The answer to each has been verified by running it with JDK 21.

Example 4.2.1

The `DigitalClock` class written with access methods is used (hour 0–23, minute and second 0–59). What do the two code fragments below print? How many `DigitalClock` objects are created in the second fragment?

```
DigitalClock t = new DigitalClock();
t.setHour(25);
t.setMinute(30);
t.setSecond(-1);
t.setSecond(59);
t.show();
```

```
DigitalClock a = new DigitalClock();
DigitalClock b = a;
a.setMinute(15);
b.setMinute(45);
b = new DigitalClock();
b.setMinute(10);
System.out.println(a.getMinute() + " " + b.getMinute());
```

```
00:30:59
45 10
```

In the first fragment `setHour(25)` and `setSecond(-1)` are rejected by the setters. The hour stays at its initial value 0; the minute is 30, and the second becomes 59 with the last valid call. The result is `00:30:59`.

In the second fragment the assignment `b = a` creates no new object; both variables point to the same object. `a.setMinute(15)` and `b.setMinute(45)` are applied to the same object and the minute becomes 45. Then `b` is attached to a new object and its minute is made 10. `a` keeps pointing to the first object. The output is `45 10`. In total two objects have been created, because `new` ran twice.

Example 4.2.2

Which of the lines A–E below give a compile error? Why?

```
public class Test {
    public static void main(String[] args) {
        DigitalClock s = new DigitalClock();
        s.setSecond(30);                    // A
        int x = s.second;                   // B
        System.out.println(s.getSecond());  // C
        s.getSecond() = 10;                 // D
        s.setSecond("40");                  // E
    }
}
```

```
Test.java:5: error: second has private access in DigitalClock
        int x = s.second;                   // B
                ^
Test.java:7: error: unexpected type
        s.getSecond() = 10;                 // D
                      ^
  required: variable
  found:    value
Test.java:8: error: incompatible types: String cannot be converted to int
        s.setSecond("40");                  // E
                    ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full
output
3 errors
```

Line B tries to read a `private` field; the right way is `s.getSecond()`. In line D an assignment to the result of a method call is attempted. A method call is a value, not a variable; the `required: variable, found: value` in the message says this. The right way is `s.setSecond(10)`. In line E `"40"` is a `String` and is not automatically converted to an `int` parameter. A and C are correct, because they call `public` methods.

Example 4.2.3

What does the following code print with the `BankAccount` class? Which balances are seen on the last line?

```
BankAccount h = new BankAccount(111, 1000.0, "Checking");
h.withdraw(5000.0);
h.deposit(-50.0);
h.addInterest(3.0);
h.withdraw(250.0);
BankAccount v = new BankAccount(222, 100.0, "Savings");
h.transfer(v, 1000.0);
h.transfer(v, 750.0);
System.out.println(h.getBalance() + " " + v.getBalance());
```

```
Insufficient balance.
Invalid deposit amount.
Interest applies only to savings accounts.
250.0 TL withdrawn. New balance: 750.0
Insufficient balance or invalid amount.
750.0 TL deposited. New balance: 850.0
750.0 TL transferred. New balance: 0.0
0.0 850.0
```

The balance is 1000.0. The first withdrawal is rejected because 5000.0 exceeds the balance. −50.0 is an invalid deposit amount. Because the account is a checking account, no interest is added. The withdrawal of 250.0 is valid and the balance drops to 750.0. The second account is a savings account with a balance of 100.0. The transfer of 1000.0 is rejected because it exceeds the balance. The transfer of 750.0 is exactly the whole balance and satisfies the condition `amount ≤ balance`. First the target account's message (850.0), then the transfer's message (0.0) is printed. The last line is `0.0 850.0`.

Example 4.2.4

(a) What does the first code below print with the non-copying `ReportCard`? What does it print with the defensively copying `ReportCard`? (b) What does the second code print with the `Money` class? (c) What happens if the `Counter` program that calls `increment()` ten times is run without `-ea` and with `-ea`? (d) Does `new Ratio(1, 2).num` compile in a class other than `Ratio`?

```
int[] d = {50, 60};
ReportCard k = new ReportCard(d);
int[] g = k.getGrades();
g[0] = 100;
d[1] = 100;
System.out.println(k.average());
```

```
Money p = new Money(100, "TL");
p.plus(p);
p = p.plus(p);
p.plus(new Money(1, "TL"));
System.out.println(p);
```

(a) In the non-copying version `d`, `g` and the field point to the same array. Both assignments reach the object: the array becomes `{100, 100}` and the average is `100.0`. In the version with defensive copies both assignments go to copies; the object's array stays `{50, 60}` and the average is `55.0`. JDK 21 gives these values in both versions.

(b) The results of the calls `p.plus(p)` and `p.plus(new Money(1, "TL"))` are discarded. Only `p = p.plus(p)` is kept: 100 + 100 = 200 kuruş. The output is `2.00 TL`. That an object takes itself as an argument is also no problem, because `plus` changes no object.

(c) Without `-ea` the `assert` line does not run and the program prints `done`; the error is not visible. With `-ea`, on the tenth increment the value becomes 10 and `AssertionError: value=10` is thrown. Only the second run shows the error.

(d) No. The compiler gives the error `num has private access in Ratio`. `private` access is free only in the code of `Ratio` itself.

Example 4.2.5

Let us write four small classes. `Student`: `private` name and grade; the grade is accepted only between 0 and 100; `passed()` returns `true` if the grade is ≥ 50. `Point`: accepts only non-negative coordinates; `distance()` returns the distance from the origin. `Thermometer`: stores a Celsius value and rejects anything below −273.15; `getFahrenheit()` computes without a separate field. `Counter`: has only the methods `increment()`, `reset()` and `getValue()`.

```
public class Student {
  private String name;
  private int grade;
  public Student(String name) { this.name = name; }
  public void setGrade(int grade) {
    if (grade >= 0 && grade <= 100) this.grade = grade;
  }
  public int getGrade() { return grade; }
  public boolean passed() { return grade >= 50; }
}
```

```
public class Point {
  private double x, y;
  public Point(double x, double y) { setX(x); setY(y); }
  public void setX(double x) { if (x >= 0) this.x = x; }
  public void setY(double y) { if (y >= 0) this.y = y; }
  public double getX() { return x; }
  public double distance() { return Math.sqrt(x * x + y * y); }
}
```

```
public class Thermometer {
  private double celsius;
  public Thermometer(double c) { setCelsius(c); }
  public void setCelsius(double c) { if (c >= -273.15) celsius = c; }
  public double getCelsius() { return celsius; }
  public double getFahrenheit() { return celsius * 9 / 5 + 32; }
}
```

```
public class Counter {
  private int value;
  public void increment() { value++; }
  public void reset() { value = 0; }
  public int getValue() { return value; }
}
```

```
public class Exercise {
  public static void main(String[] args) {
    Student o = new Student("Ece");
    o.setGrade(120);
    System.out.println(o.getGrade() + " " + o.passed());
    o.setGrade(75);
    System.out.println(o.getGrade() + " " + o.passed());
    Point n = new Point(3, 4);
    System.out.println(n.distance());
    n.setX(-7);
    System.out.println(n.getX());
    Thermometer t = new Thermometer(25);
    System.out.println(t.getFahrenheit());
    t.setCelsius(-300);
    System.out.println(t.getCelsius());
    Counter s = new Counter();
    s.increment(); s.increment(); s.increment();
    System.out.print(s.getValue() + " ");
    s.reset();
    System.out.println(s.getValue());
  }
}
```

```
0 false
75 true
5.0
3.0
77.0
25.0
3 0
```

`setGrade(120)` is rejected and the grade stays at 0: `0 false`. After `setGrade(75)`, `75 true` is printed. The constructor of `Point` calls the setters; the rule thus stays in one place. For `new Point(3, 4)` the distance is $\sqrt{(9 + 16)} = 5.0$. `setX(-7)` is rejected and x stays 3.0. 25 °C is 25 × 9 / 5 + 32 = 77.0 °F. The derived value is not kept in a separate field; had it been, the two fields would have to be kept consistent on every change. After `setCelsius(-300)` the value stays 25.0.

`Counter` has no setter, because the value of the counter should only increase one by one or be reset. A `setValue` would break this rule. Three `increment()` calls make the value 3; `reset()` returns it to 0.

A small note on the constructor calling setters: when compiled with `javac -Xlint:all`, JDK 21 gives the warning `possible 'this' escape` for `Point` and `Thermometer`. The warning says that a subclass could override these methods and see a half-built object. If the class is made `final`, or the methods called are `final` or `private`, the warning disappears.

Example 4.2.6

(a) The `Student` class is to be moved to the `school` package and used from `app.Main`. Which lines are added, and what are the compile and run commands? (b) An `opCount` field with only a getter is to be added to `BankAccount`; it increases by one on every successful deposit and withdrawal. At the end of the two-account trace example, what is the operation count of the two accounts?

(a) `package school;` is written at the top of `Student.java`, and `package app;` and `import school.Student;` at the top of `Main.java`. The files are put in the folders `src/school/` and `src/app/`. The commands: `javac -d out src/school/Student.java src/app/Main.java` and `java -cp out app.Main`. The `Student` class and the methods used must be `public`.

```
1 2
```

(b) The counter is incremented in `deposit` and `withdraw`, only on the successful branches. The counter of `acc1` is 1: only `deposit(500.0)` counts. `addInterest` is not a deposit; `transfer` deducts directly from its own balance and does not call its own `withdraw`. The counter of `acc2` is 2: `withdraw(200.0)` and the `deposit` inside the transfer. This result also shows a side effect of the transfer not doing its own side of the operation through `withdraw`. If the rules are to be gathered in one place, `transfer` should call `withdraw` too.

4.2.10 Common mistakes

The compiler catches some of the mistakes related to encapsulation and points to their location with its message. The others lead to programs that compile but silently corrupt the object. Those in the second group are more dangerous.

Common mistake

Accessing a `private` field from outside. Message: `second has private access in DigitalClock`. Instead of `s.second = 75;`, write `s.setSecond(75);`; to read, use `s.getSecond()`.

Common mistake

Forgetting `this` in a setter. `width = width;` compiles but assigns the parameter to itself; the field does not change and `getWidth()` returns 0. The correct form is `this.width = width;`.

Common mistake

Writing a getter as `void`. The line `public void getWidth() { return width; }` gives the error `incompatible types: unexpected return value`. A `void` method cannot return a value; the return type must be the type of the field, that is, `int`.

Common mistake

Writing a setter without validation. `public void setSecond(int s) { second = s; }` is no different from making the field `public`. The validity rule is checked in the setter.

Common mistake

Writing a method name without parentheses. In the line `int d = s.getMinute;` the compiler looks for a field called `getMinute`: `cannot find symbol ... variable getMinute`. A method call is always written with parentheses.

Common mistake

Not writing an interface method. Message: `SportsCar is not abstract and does not override abstract method stop() in Car`. A class that says `implements` must write all the methods in the interface.

Common mistake

Handing out the inner object through a getter. `return grades;` exposes a mutable array or list to the outside. A copy or a read-only view is returned. The array that comes into the constructor is copied too.

Common mistake

Thinking a `final` field is immutable. When `private final int[] grades` is written, what is fixed is the reference, not the elements of the array. `grades[0] = 0;` compiles.

Common mistake

Not validating in the constructor. Even if the setters check, `new BankAccount(1, -500.0, ...)` builds an invalid object. The constructor must enforce the rule too.

Common mistake

Forgetting `import`, or thinking a subpackage is imported. Message: `cannot find symbol ... class Scanner`. After `import java.util.*;` the same error appears for `Function`; `java.util.function` is imported separately.

Common mistake

Using a package-private class from outside. Message: `Helper is not public in bank; cannot be accessed from outside package`. The class is either made `public` or used only from the same package. The methods used must be `public` too.

Common mistake

Writing a setter for every field. The rule slides over to the code that uses the class. Behavior (`withdraw`, `addBonus`) is designed first; most of the time no setter is needed.

Common mistake

Swallowing the error silently. Ignoring an invalid value without leaving any trace hides the error. The behavior is either stated explicitly in the contract or an exception is thrown.

4.2.11 Summary

- **Encapsulation** is gathering the data and the methods that process it in a class and closing the data off from the outside. In Java, fields are written `private` and behaviors `public`.
- **Information hiding** separates the "what" of a task from its "how". The user knows only the method signature and its contract; the user does not know the steps behind the call `service.addData("Ayşe")`.
- A **precondition** is the state assumed true before the call, a **postcondition** the state guaranteed after the call. Together they are the method's contract and are written in Javadoc with the tags `@param`, `@return` and `@throws`. The `<` signs in a comment are put inside `{@code ...}`.
- When the precondition is broken, the method may ignore the value, return a status or throw an exception. For `public` methods the customary choice is `IllegalArgumentException`.
- A **class invariant** is a rule every object satisfies at every moment. `assert` checks the invariant during development, but works only with `-ea`; it is not used for argument checking.

- There are four **access levels**: `public` (everywhere), `protected` (package and subclasses), package access (package only) and `private` (class only). The most restrictive level is preferred. `private` access is checked per class, not per object.
- A **getter** returns the field, a **setter** checks the new value and assigns it. Because the parameter shadows the field, the assignment is written as `this.field = field;`.
- The constructor must validate too. If a constructor throws an exception, `new` produces no reference and the invalid object never reaches the user.
- The **interface** is the outer face of a class and the **implementation** its internal mechanism. `interface` and `implements` turn this distinction into a construct the compiler checks. Even when the implementation changes, the interface can stay the same: the two-field and the single-field clock give the same output.
- "Tell, don't ask": the object that holds the data makes the decision; a setter is written only if the field can really change freely.
- The reference to a mutable inner object can **leak**. A **defensive copy** is made in the constructor and the getter, or a read-only view is given. `final` fixes only the reference.
- An **immutable class** has no setters, its fields are `private final`, the class is `final`, and "changing" methods return a new object. Forgetting to assign the result (`a.plus(p);`) is the most frequent mistake.
- A **package** is a group of classes in a folder; the full name of a class includes its package. A packaged program is compiled with `javac -d out ...` and run with `java -cp out pkg.Class`. `import` lets a class from another package be used by its short name; it does not bring in subpackages. Of two classes with the same name, one is imported and the other is written with its full name.

4.2.12 Key terms

encapsulation information hiding data hiding abstraction precondition / postcondition class invariant assertion exception access modifier accessor (getter) mutator (setter) shadowing this keyword constructor field reference interface interface vs. implementation implement defensive copy immutable record package import

---

4.2.13 Test Yourself

1. For `DigitalClock` written with access methods, this code is run: `DigitalClock a = new DigitalClock(); DigitalClock b = a; a.setHour(23); b.setHour(a.getHour() + 1);`

`a.setMinute(a.getMinute() - 1); b.setSecond(59); a.show();`. What is printed?

Answer

`a` and `b` point to the same object. The hour becomes 23. `b.setHour(24)` is rejected, because valid hours are 0–23. `setMinute(-1)` is rejected too and the minute stays at 0. The second becomes 59. Output: `23:00:59`.

2. The `public class A` in package `p1` has the fields `public int pub`, `protected int prot`, `int pkg` with no modifier and `private int priv`. In package `p2`, a method of the class `C extends p1.A` assigns a value to all four fields by their own names. Which assignments do not compile?

Answer

The assignments to `pub` and `prot` compile: `public` is accessible from everywhere and `protected` from a subclass in another package. For `pkg` the error `pkg is not public in A; cannot be accessed from outside package` appears, and for `priv` the error `priv has private access in A`.

3. A single-field (`totalSeconds`) `DigitalClock` has just been created. `setMinute(59)`, `setSecond(59)` and `setMinute(0)` are called in turn. What is `totalSeconds` after each call? What does `getMinute() + ":" + getSecond()` give at the end?

Answer

In the first call the difference is 59 minutes: 59 · 60 = 3540. In the second `getSecond()` is 0 and 59 is added: 3599. In the third `getMinute()` is 3599 / 60 % 60 = 59; the difference is −59 minutes, that is, −3540 seconds: 59. The result is `0:59`. The second part is not affected by the change of minute.

4. The condition in the `addData` method has been written by mistake as `if (data.isEmpty() || data == null)`. What happens on the call `addData(null)`? If the condition is made `data == null || data.trim().isEmpty()`, how many of the arguments `" "`, `null` and `" a b "` add a record?

Answer

In the first form `isEmpty()` is called on `null` and the program stops with a `NullPointerException`; the short circuit of `||` helps only if the left side is true. In the second form `" "` is empty after trimming and is rejected, `null` is rejected in the first condition; only `" a b "` is added. JDK 21 output: `false false true 1`.

5. In a Javadoc comment, `@param n must satisfy 0 < n < 10` has been written. What does the

`javadoc` command do? How is the comment fixed?

> **Answer**
>
> Javadoc processes the comment as HTML and takes a raw `<` sign to be the start of a tag. JDK 21 gives a `malformed HTML` error. The fix: `@param n must satisfy {@code 0 < n < 10}`. The text inside `{@code ...}` is shown as it is. Writing `<` also works, but it is hard to read in the source.

6. From the list `List<Integer> inner = new ArrayList<>(List.of(70, 80));`, `view = Collections.unmodifiableList(inner)` and `copy = new ArrayList<>(inner)` are produced; then `inner.add(90)` runs. What do `view` and `copy` contain? What does `view.add(100)` do?

> **Answer**
>
> The view does not copy; it shows the inner list: `[70, 80, 90]`. The copy is independent: `[70, 80]`. `view.add(100)` throws an `UnsupportedOperationException`. The view prevents changes from outside but reflects the changes made inside the class itself.

7. With the `Money` class, what does `System.out.println(new Money(-50, "TL"));` print? Is this a bug; what rule can be added to the class?

> **Answer**
>
> Because −50 / 100 = 0 and −50 % 100 = −50, `0.-50 TL` is printed. The amount looks meaningless. If the class will not allow negative amounts, the constructor should throw an `IllegalArgumentException` for `cents < 0`. If it will allow them, `toString` should write the sign separately and work with the absolute value.

8. `javac -d out src/bank/Account.java src/app/Main.java` was run in the project folder. Then `java -cp out/app Main` was tried. What happens, and why?

> **Answer**
>
> Java finds the file `out/app/Main.class`, but the full name of the class in the file is `app.Main`. Result: `NoClassDefFoundError: Main (wrong name: app/Main)`. The right command is `java -cp out app.Main`: the classpath is the root of the packages and the class name is the full name.

9. The constructor of the `Product` class assigns the price with `setPrice`; `setPrice` silently ignores a non-positive value. What does `new Product("Pen", -5).getPrice()` return? What is the

problem with this design?

Answer

No assignment is made and the field stays at its default value `0.0`. The object has been built, but a product with price 0 breaks the invariant "positive price only". Silent rejection does not work in a constructor. The constructor should throw an `IllegalArgumentException` for an invalid value, so that the object is never built.

4.2.14 Further reading

- J. Bloch, *Effective Java* (3rd ed., Addison-Wesley, 2018), Items 15–17. Explains, in short rules with reasons, minimizing the accessibility of classes and members, minimizing mutability, and using accessor methods instead of `public` fields.
- Oracle, *The Java Tutorials*, "Controlling Access to Members of a Class" and "Creating and Using Packages": explain the access levels and the compilation of packaged programs with official examples.
- A. B. Downey and C. Mayfield, *Think Java* (2nd ed., O'Reilly, 2019), Chapter 11 "Designing Classes": builds a class step by step with its fields, constructors, access methods and invariants; it can be read free online.
- J. Gosling et al., *The Java Language Specification*, Java SE 21, §6.6 and §7: the precise rules of access control and packages; the reference to consult especially for the details of `protected` access.

4.2.15 References

- Oracle, *The Java Tutorials*: "Controlling Access to Members of a Class", "Using the this Keyword", "Packages", "Interfaces".
- J. Gosling et al., *The Java Language Specification*, Java SE 21 Edition (6.6 Access Control, 7 Packages and Modules, 7.5 Import Declarations, 15.12.2 method selection).
- P. Deitel and H. Deitel, *Java: How to Program, Early Objects*, 11th ed., Pearson, 2017 (Chapters 3 and 8: Classes and Objects).
- K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly, 2022 (Chapter 4: encapsulation).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., 2022 (Chapter 4: Objects and Classes).
- J. Bloch, *Effective Java*, 3rd ed., Addison-Wesley, 2018 (Items 15–16: minimizing accessibility, accessor methods instead of `public` fields).
- A. B. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019 (Chapter 11: Designing Classes).

PART II · CHAPTER 4.3

Inheritance

Real programs are full of classes that look alike. A student and a teacher both have a name and an age; a cat, a dog and a bird all eat, sleep and make a sound. Rewriting the shared part in every class costs effort and invites bugs: a fix made in one place is forgotten in another. Inheritance solves this. The shared fields and methods are written once, in a superclass; subclasses receive them by inheritance and add only what makes them different. These notes first explain what inheritance is, the `extends` syntax, and what a subclass receives. They then follow the `super` keyword, the constructor chain and the order of initialization step by step. Method overriding and its rules, `protected` access, and limiting inheritance with `final` come next. The last sections cover the is-a and has-a relationships between classes, the design traps of inheritance, and the `toString`, `equals` and `hashCode` methods of `Object`, the ancestor of every class.

LEARNING OUTCOMES

Concept
Define inheritance, superclass and subclass; recognize the is-a relationship.

Syntax
Write a subclass with `extends`; explain what is inherited and what is not.

Constructors
Use `super(...)` and `super.method()`; trace the order in which constructors and initialization run.

Overriding
Override a method, tell overriding from overloading, use `@Override`, and explain the reasons behind the rules.

Access and restriction
Explain `protected` access and how `final` restricts inheritance.

Design
Choose between inheritance and composition; recognize the fragile base class problem and violations of the substitution principle.

The Object class
Write `toString`, `equals` and `hashCode` correctly; know the difference between `==` and `equals`.

4.3.1 The concept of inheritance

A class is a template for objects of the same structure: it says what data they hold and what they can do. In a real program, classes do not stand apart. Some are more specific forms of others. Every rectangle is a shape, every student is a person, every cat is an animal. Inheritance turns this "more specific form of" relationship between classes into code. This section defines inheritance, gives the syntax in Java, explains what a subclass receives, and shows how a subclass object is laid out in memory.

4.3.1.1 What is inheritance?

Definition 4.3.1

Inheritance is creating a new class by taking the fields and methods of an existing class and enriching them with new or modified capabilities. The existing class is called the **superclass**, the new class the **subclass**.

Two verbs in the definition matter: *taking* and *enriching*. A subclass takes the fields and methods of its superclass without rewriting them. The inherited members are not defined again in the subclass; they are used as if they were its own. On top of these, the subclass can add new fields and new methods. It can also change the behavior of a method it took from the superclass.

An everyday sentence sums up the relationship: "Every rectangle is a shape." In a drawing program every shape has a color and every shape can be drawn. A rectangle has all of these common properties. On top of them it has its own width and height, and its area is computed from the two. The relationship can also be shown in a diagram. Figure 4.3.1 shows it as a UML (Unified Modeling Language) class diagram.

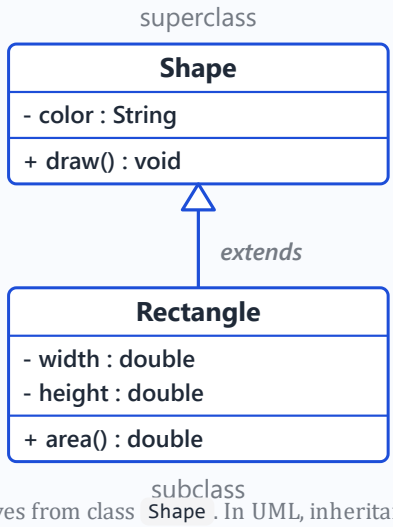


Figure 4.3.1: Class `Rectangle` derives from class `Shape`. In UML, inheritance is drawn as an arrow with a hollow triangle head pointing at the superclass. The subclass box lists only the members the subclass adds itself; `color` and `draw()` arrive by inheritance.

In the diagram, the `Rectangle` box has no `color` field and no `draw()` method. They have not been written there, yet every `Rectangle` object has them. That is exactly what inheritance means: the shared part is written once, in the superclass, and each subclass carries only its own difference. A reader following the arrow forms the sentence "a `Rectangle` is a `Shape`". The direction of the arrow is the direction of that sentence.

4.3.1.2 A subclass with `extends`

In Java, the fact that a class derives from another class is declared with the `extends` keyword. The syntax is `class SubClass extends SuperClass`. The subclass extends the superclass; that is what the word means.

```
// base class (superclass)
public class Shape {
    // fields and methods of the base class
    // ...
}

// new class (subclass), derived from Shape
public class Rectangle extends Shape {
    // fields and methods of the new class
    // ...
}
```

The bodies of these two classes are still empty, but the relationship is established. Every object of type `Rectangle` is now a `Shape`. Every `public` and `protected` member later added to `Shape` can be used in `Rectangle` as well. The subclass code calls these members as if they were its own, with no

object name or prefix.

Only **one** class may follow `extends`. Java does not allow multiple inheritance of classes. A class that never writes `extends` still has a superclass: the compiler quietly derives it from `java.lang.Object`. Figure 4.3.2 shows this two-step chain.

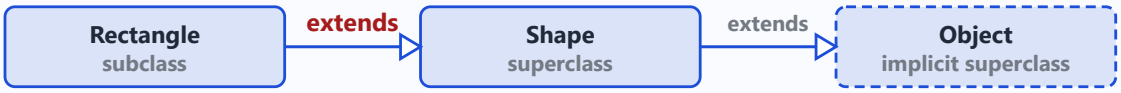


Figure 4.3.2: `Rectangle` extends `Shape` explicitly. Because `Shape` does not write `extends`, the compiler derives it from `Object`. The chain always ends at `Object`.

There is one more thing a subclass should not do. If a field with the same name as a superclass field is declared again in the subclass, Java does not complain. But the two fields are not the same variable. The object holds two separate variables; the subclass code sees its own field and the superclass code sees its own. This is called **field hiding**. It is hard to follow which code reads which value, and the bugs appear silently. A field that arrives by inheritance is not declared again in the subclass.

Common mistake

Adding `String color` to the subclass when the superclass already has `String color`. The subclass constructor writes `color = "red"`, but the superclass's `draw()` method reads its own `color` field and sees `null`. The right way is to keep the field only in the superclass and set its value through the superclass constructor.

4.3.1.3 Subclass and superclass: specialization

A subclass exhibits all the behaviors of its superclass. In addition, it can exhibit behaviors of its own. A cat does everything an animal does: it eats, sleeps and breathes. It also meows and climbs. This is why inheritance is also called **specialization**. The subclass represents a more specific group within the group the superclass represents.

The relationship is one-directional. Every cat is an animal, but not every animal is a cat. In code the consequence is clear. A `Cat` object can be assigned to a variable of type `Animal`, because a cat can do everything expected of an animal. The reverse is not allowed. A plain `Animal` object cannot be assigned to a `Cat` variable; nothing says that object can meow.

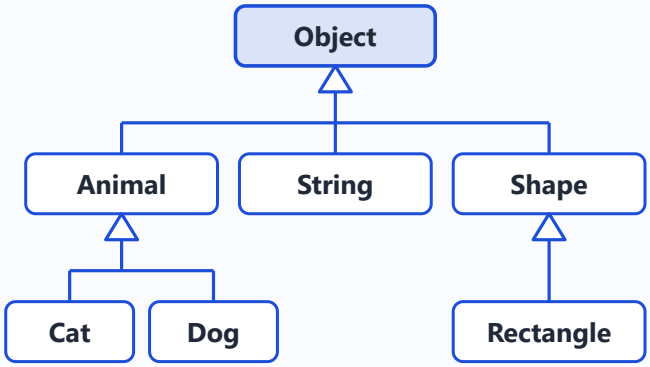


Figure 4.3.3: A small part of the Java class hierarchy. `Object` is at the top. Going up, classes become more general; going down, more specific.

Java's class hierarchy starts with the `Object` class in package `java.lang`. Figure 4.3.3 shows a slice of it. `Animal`, `String` and `Shape` derive directly from `Object`; `Cat` and `Dog` derive from it indirectly, through `Animal`. Every class inherits from `Object`, directly or indirectly. That is why the methods of `Object` are available on every object.

4.3.1.4 Kinds of inheritance

Inheritance relationships between classes can be arranged in four ways. Figure 4.3.4 shows all four side by side.

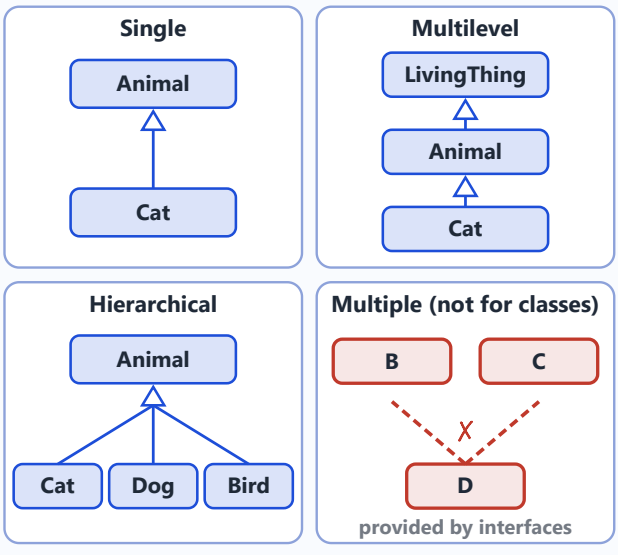


Figure 4.3.4: Kinds of inheritance. Single, multilevel and hierarchical inheritance exist in Java. Multiple inheritance, where a class has two superclasses, does not exist for classes; interfaces meet the same need.

- In **single** inheritance there is one subclass and its one superclass: `Cat extends Animal`.

- In **multilevel** inheritance the classes form a chain: `Animal extends LivingThing` and `Cat extends Animal`. `Cat` receives the members of `LivingThing` through `Animal`. Each level carries everything from the one before.
- In **hierarchical** inheritance many subclasses derive from one superclass: `Cat`, `Dog` and `Bird` all derive from `Animal`. The shared code is written once; each subclass adds only its own difference.
- In **multiple** inheritance a class has more than one superclass. Java does not allow this for classes. Interfaces meet the same need: a class can extend a single class while implementing many interfaces.

As a consequence of these rules, the class hierarchy in Java is a **tree**. The root of the tree is `Object`. Every class other than `Object` has exactly one direct superclass. There is no limit on the depth of a chain. But deep chains make code harder to read and change: to understand the behavior of one class, you must look at all the classes above it. In practice, hierarchies deeper than two or three levels are built only for good reason.

4.3.1.5 What is inherited?

The sentence "a subclass takes everything from its superclass" is only partly true. The true part is this: *inside* a subclass object, all the fields of the superclass are present. But whether the code of the subclass may reach those fields and methods directly is decided by the access modifier. Constructors are never inherited. Table 4.3.1 summarizes the situation by kind of member.

Member	Passed to the subclass?	Note
`public` / `protected` fields and methods	Yes	Used directly in the subclass.
Members with no modifier (package access)	Only if in the same package	A subclass in another package cannot access them.
`private` fields	Present in the object, but not directly accessible	Used through the superclass's `public` / `protected` methods (getters, setters).
`private` methods	No	Invisible; cannot be overridden.
Constructors	No	A subclass constructor calls the superclass's with `super(...)`.
`static` members	Accessible	They belong to the class; they are not overridden, only hidden.

Table 4.3.1: Are the members of the superclass passed on to the subclass?

The row of the table that is most often confused is the one for `private` fields. Suppose `Animal` has a `private double health` field. When `Cat extends Animal` is written, every `Cat` object contains a `health` field. When the cat sleeps, its health rises and the value is stored in this field. But if the code of class `Cat` writes `health += 5`, the compiler reports an error with the message `health has private access in Animal`. The field is there, but access by name is open only to the code of `Animal` itself. `Cat` reaches the field through the methods `Animal` offers.

There is a good reason constructors are not inherited either. A constructor's name is the name of its class. If the constructor `Animal(String name)` were passed on to `Cat`, the call `new Cat("Mavi")` would set up only the `Animal` part, and `Cat`'s own fields would be left unset. So every class writes its own constructor. The subclass constructor calls the superclass constructor with `super(...)` so that the superclass part gets set up.

Example 4.3.1

Let class `Animal` have these members: `private String name`, `protected int age`, a package-private `void eat()`, `public String getName()` and `private void digest()`. `Cat extends Animal` is written in another package.

In the code of `Cat`, `age` (`protected`) and `getName()` (`public`) are used directly. `name` exists in the object, but it can be reached only through `getName()`. `eat()` is package-private, and since `Cat` is in another package, it is invisible. `digest()` is `private` and never visible. Had `Cat` been a class in the same package, `eat()` could have been used too.

4.3.1.6 The structure of a subclass object

People who are new to inheritance often form this picture: writing `new Cat(...)` creates a `Cat` object and, inside it or next to it, an `Animal` object. That picture is wrong. `new Cat(...)` creates **a single object**. No separate object is created for the superclass.

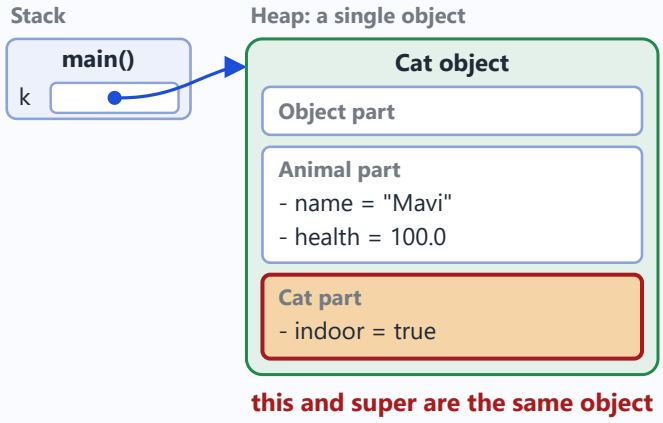


Figure 4.3.5: A `Cat` object. The `Object`, `Animal` and `Cat` parts are inside the same object. The variable `k` on the stack points to this single object. `this` and `super` look at the same object.

Figure 4.3.5 shows the inside of the object. The object is made of layers. The innermost layer is the part that comes from `Object`. Above it sit the fields of `Animal` (`name`, `health`), and on top is `Cat`'s own field (`indoor`). Even when the fields of `Animal` are `private`, they are inside this object. Only the code of `Cat` cannot reach them by name.

In this picture the `super` keyword does not point to a separate object. `super` is a way of reaching the superclass part of the same object through the superclass's code. `this` and `super` carry the same address; the difference lies in which class's method gets called. This distinction is the key to understanding the constructor chain and overriding.

There is a proof of this. The `getClass()` method of `Object` returns the actual runtime class of the object. In the following program the `getClass()` call is inside the constructor of `Animal`:

```
class Animal {
    Animal() {
        System.out.println("In Animal(): "
            + getClass().getSimpleName());
    }
}
class Cat extends Animal { }
// in main:
new Cat();
new Animal();
```

```
In Animal(): Cat
In Animal(): Animal
```

The first line may look surprising. The code that runs is the constructor of `Animal`, yet the object's class comes out as `Cat`. The reason is this: `new Cat()` first reserves space for a `Cat` object and then sets up the parts of this object in order. While `Animal()` runs, what it sets up is the `Animal` part of a `Cat`. There is no separate `Animal` object. On the second line, `new Animal()` really does create an `Animal` object. No constructor is written in class `Cat`; the compiler adds a no-argument default constructor to it, and this constructor calls `Animal()`.

4.3.1.7 Moving shared code up to the superclass

Inheritance is often not set up at the start of a design. A few classes are written first, and later repeated code is noticed among them. Finding this repetition and moving it into a common superclass is called **generalization**. Generalization is the reverse of specialization. In specialization you go down from a general class to a specific one; in generalization you go up from specific classes to a common general one.

Suppose a school management program starts with these two classes:

```
class Student {
    private String name;
    private int age;
    private double gpa;
    String info() { return name + " (" + age + ")"; }
}
class Teacher {
    private String name;
    private int age;
    private String subject;
    String info() { return name + " (" + age + ")"; }
}
```

In both classes the fields `name` and `age` and the method `info()` are exactly the same. The price of this repetition is paid when something changes. Say the display format of the age changes, for example “20 years old” instead of “(20)”. Both classes must be fixed. If one is forgotten, the program prints the same information in two different formats. The more classes there are, the more likely a class is forgotten.

Once the shared part is moved to a `Person` class, the code looks like this:

```
class Person {
    private String name;
    private int age;
    Person(String name, int age) { this.name = name; this.age = age; }
    String info() { return name + " (" + age + ")"; }
}
class Student extends Person {
    private double gpa;
    Student(String name, int age, double gpa) {
        super(name, age);
        this.gpa = gpa;
    }
    @Override String info() {
        return super.info() + ", gpa " + gpa;
    }
}   // Teacher likewise: adds subject
```

`Student` now carries only its own difference: the `gpa` field. It does not store the name and age itself; in its constructor it passes them to the constructor of `Person` with `super(name, age)`. It overrides `info()` but does not rewrite the shared part: with `super.info()` it calls the version of `Person` and appends the grade average. `Teacher` is written the same way: its field is `subject`, and it appends the subject to the end of `info()`. When the program runs, two lines are printed:

```
Ece (20), gpa 3.4
Murat (41), Physics
```

Now the display format of the age changes in a single place, `Person.info()`; both subclasses use the new format automatically.

Before generalizing, ask a question: “Is every student a person?” If the answer is yes, generalization is right. If only a few fields of two classes look alike and there is no “is a kind of” relationship between them, inheritance is the wrong tool. For example, a `Student` and a `Course` may both have a `name` field. But a course is not a student. In such cases the shared code goes into a separate helper class and is used through composition.

Example 4.3.2

Suppose an electric circuit simulation program has classes `Resistor`, `Capacitor` and `Inductor`. In all three, two node numbers (`nodeA`, `nodeB`), a name and a method `connect(int a, int b)` are repeated. Each one's own value differs: resistance, capacitance and inductance.

The answer to “is every resistor a circuit element?” is yes, and the same holds for the other two. The shared fields and the `connect` method move up into a `CircuitElement` superclass. Each subclass keeps only its own value and the computation that depends on it, for example an `impedance(double frequency)` method. If the checking of node numbers changes, only `CircuitElement` needs fixing.

4.3.2 The animal example

The parts of inheritance are easier to understand when they are seen together in one example. This section builds an animal hierarchy. The shared members are collected in a superclass; the subclasses change one behavior and add behavior of their own.

4.3.2.1 The animal class hierarchy

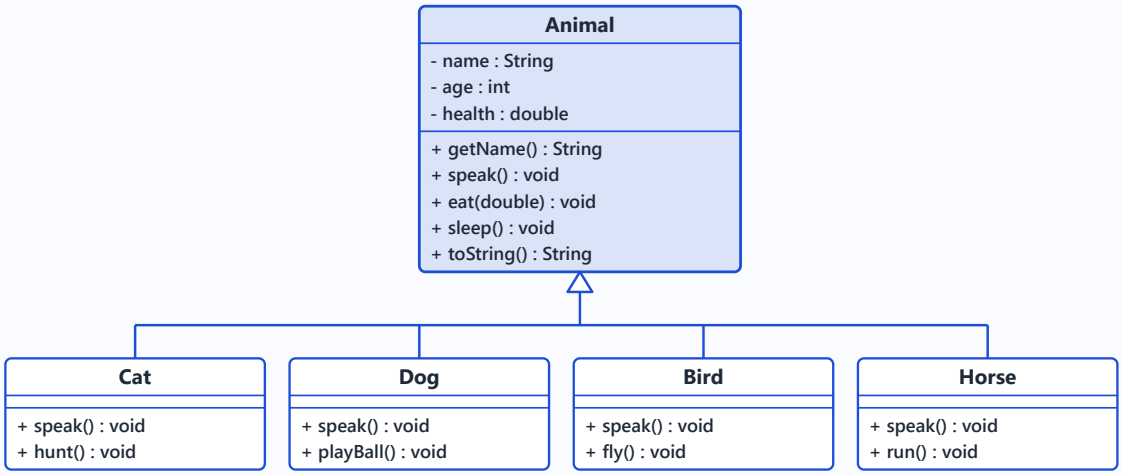


Figure 4.3.6: The classes `Cat`, `Dog`, `Bird` and `Horse` derive from `Animal`. Each overrides `speak()` in its own way and adds a method of its own. The `-` sign marks a `private` member, the `+` sign a `public` one.

Figure 4.3.6 is a typical example of hierarchical inheritance. Class `Animal` carries the fields common to all animals: `name`, `age`, `health`. The common behaviors are written in it too: `speak`, `eat`, `sleep`, `toString`. These members are written once, in `Animal`. The boxes of the four subclasses hold only two methods. One is `speak()`; since every animal makes a different sound, each subclass rewrites this method its own way. The other is a behavior specific to that animal: the cat hunts, the dog plays ball, the bird flies, the horse runs.

The diagram yields one more conclusion. Adding a new kind of animal does not require touching `Animal`. For example, a `Cow` class derives from `Animal`, overrides `speak()` and adds a `milk()` method. Sleeping, eating and health tracking come for free.

4.3.2.2 Classes `Animal` and `Dog`

The code of the hierarchy starts with two classes. The superclass `Animal` holds the fields and behaviors common to all animals:

```
public class Animal {
    private String name;     // common fields
    private int age;
    private double health;

    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
        this.health = 100.0;
    }
    public String getName() { return name; }   // access to private name
    public void speak() {
        System.out.println("This animal speaks.");
    }
    public void sleep() {
        health += 5;
        System.out.println(name + " slept. Health: " + health);
    }
}
```

All three fields are `private`. They cannot be reached directly from outside or from subclasses. The way to reach `name` is the `getName()` method. The constructor takes two values and starts health at 100. `speak()` prints a generic message; how a particular animal speaks is not yet known. `sleep()` raises health by 5 and prints the state.

The subclass `Dog` writes only the difference:

```
public class Dog extends Animal {
    public Dog(String name, int age) {
        super(name, age);   // superclass sets the fields
    }
    @Override
    public void speak() {   // changes inherited behavior
        System.out.println(getName() + " barks.");
    }
    public void playBall() { // behavior only Dog has
        System.out.println(getName() + " is playing ball!");
    }
}
```

The class does three things. Its constructor passes the name and age it receives to the constructor of `Animal` with `super(name, age)`; `Animal` fills in the fields. It overrides `speak()`: the `@Override` annotation states that this method rewrites a method that came from the superclass. Finally it adds the method `playBall()`, which only dogs have. Note that the code of `Dog` never touches the `name` field directly; it gets the name with `getName()`. Since `name` is `private`, there is no other way.

```
Dog dog = new Dog("Karabaş", 4);
dog.speak();     // Dog's version
dog.playBall();   // Dog only
dog.sleep();       // inherited from Animal
```

```
Karabaş barks.
Karabaş is playing ball!
Karabaş slept. Health: 105.0
```

The three calls show three different situations. `dog.speak()` runs the version that `Dog` overrides; "This animal speaks." is not printed. `dog.playBall()` is a method only `Dog` has. `dog.sleep()` is not written in `Dog` at all. Java looks for this method in `Dog` first, and when it does not find it, goes up one level to `Animal` and finds it there. The method is the code of `Animal`, yet it raises the `health` field of this dog object from 100 to 105.

The method lookup rule is the basic mechanism of inheritance. When a method is called on an object, the search starts at the object's *actual* class. If the method is not there, the search climbs the hierarchy. The first version found runs. A method that is not found even in `Object` is rejected by the compiler anyway.

4.3.2.3 The advantages of inheritance

Inheritance has three benefits.

- **Reuse.** The methods and fields of the superclass are used in the subclass without being rewritten. The shared code lives in one place; a fix reaches all subclasses.
- **Extension.** An existing class is extended with new capabilities without changing its code. The superclass may belong to another team or to a library; there is no need to touch its source.
- **Modification.** The methods of the superclass can be changed by overriding them in the subclass. The subclass rewrites only the part of the shared behavior that does not suit it.

A calculator gives an example of extension. Suppose we have a class `Calculator` that does the four operations and keeps the last result in a memory field. The slides show this class only as a diagram; it can be written like this:

```
class Calculator {
    private double memory;
    public double add(int a, int b) { memory = a + b; return memory; }
    public double multiply(int a, int b) { memory = a * b; return memory; }
    public double showMemory() { return memory; }
}
```

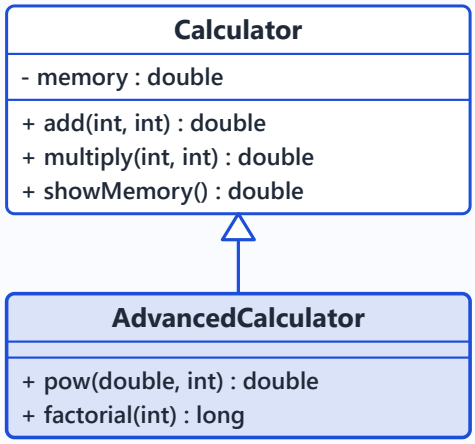


Figure 4.3.7: `AdvancedCalculator` derives from `Calculator` and adds the methods `pow` and `factorial`. The code of the superclass does not change.

Suppose a user who needs scientific calculation wants exponentiation and factorial added. Figure 4.3.7 shows the solution. The new class extends the old one; the four operations and the memory come for free:

```
public class AdvancedCalculator
        extends Calculator {
    // new ability: exponentiation
    public double pow(double base, int pow) {
        double result = 1.0;
        for (int i = 0; i < pow; i++) {
            result *= base;
        }
        return result;
    }
    // factorial(int) is added likewise
}
// in main
AdvancedCalculator h = new AdvancedCalculator();
System.out.println(h.add(5, 3));         // 8.0  (inherited)
System.out.println(h.showMemory()); // 8.0  (inherited)
System.out.println(h.pow(2, 5));            // 32.0 (new)
```

```
8.0
8.0
32.0
```

The calls `h.add(5, 3)` and `h.showMemory()` are not written in `AdvancedCalculator`; both come by inheritance and give 8.0. `pow(2, 5)` is the new method. The loop runs five times and `result` becomes 2, 4, 8, 16, 32 in turn. It is no problem in Java that the parameter `pow` has the same name as

the method: method names and variable names live in separate namespaces. For readability, it is still better to choose different names.

In terms of cost, calling a method that arrives by inheritance costs the same as calling a method of the class itself. The Java virtual machine keeps a method table for each class. A lookup amounts to reading one entry in the table; the depth of the hierarchy is not rescanned at every call. So the real price of inheritance is paid not at run time but in design. That price is tight coupling between classes.

4.3.3 Constructors and `super`

When a subclass object is built, its superclass part must be built as well. Giving initial values to the superclass's fields is the superclass's job. This section describes the tool for this cooperation between subclass and superclass, the `super` keyword. It then shows the order in which constructors run, step by step, with three code traces.

4.3.3.1 The `super` keyword

Definition 4.3.2

`super` is the keyword that refers to the superclass inside the subclass's code. It is used in two forms. `super(...)` calls the superclass constructor from inside the subclass constructor. `super.method()` calls a version of a method in the superclass; it is usually used from inside an overridden method.

There are two rules about `super(...)`. First, `super(...)` must be the **first statement** of the constructor. This rule holds in all widely used versions, including Java 21. Java 25 relaxes it a little: statements that do not touch the object under construction may appear before `super(...)`. For example, the parameters can be checked. But these notes take Java 21 as the baseline. Second, if a constructor writes neither `super(...)` nor `this(...)`, the compiler quietly inserts `super();`. So a superclass constructor is *always* called; the programmer's only choice is which one.

A design principle lies behind these rules: each class manages only its own data. `Cat` does not assign `Animal`'s `name` field itself; it leaves the job to `super(name)`. `Animal` is the class that knows best how its own fields should be initialized. Perhaps it checks that the name is not empty, perhaps it starts health at 100. The subclass need not know these rules.

```
class Cat extends Animal {
    private boolean indoor;

    Cat(String name, boolean indoor) {
        super(name);         // Animal(String)
        this.indoor = indoor;  // its own data
    }

    @Override
    public String toString() {
        return super.toString()   // Animal's
               + ", Indoor: " + indoor;
    }
}
// in main
System.out.println(new Cat("Mavi", true));
```

```
Name: Mavi, Health: 100.0, Indoor: true
```

Here the constructor `Animal(String name)` writes the name to its field and sets health to 100.0. The `toString()` method of `Animal` returns the name and health as "Name: Mavi, Health: 100.0". The constructor of `Cat` calls this constructor with `super(name)` on its first line, then fills its own field `indoor`. The `toString()` of `Cat` does not reproduce the text the superclass makes. It gets that text with `super.toString()` and appends only its own field. `println` calls the `toString()` method of the object it is given. Since the object is a `Cat`, the version in `Cat` runs.

This pattern is very common: the overriding method first has the superclass do its work with `super`, then adds its own difference. That way, when the behavior of the superclass changes, for example when `Animal.toString()` starts printing the age as well, the output of `Cat` updates by itself.

4.3.3.2 Code trace: the `super.introduce()` chain

Calls of `super.method()` can also be chained. In a three-level hierarchy, let each class override `introduce()` and let each call its superclass's version first:

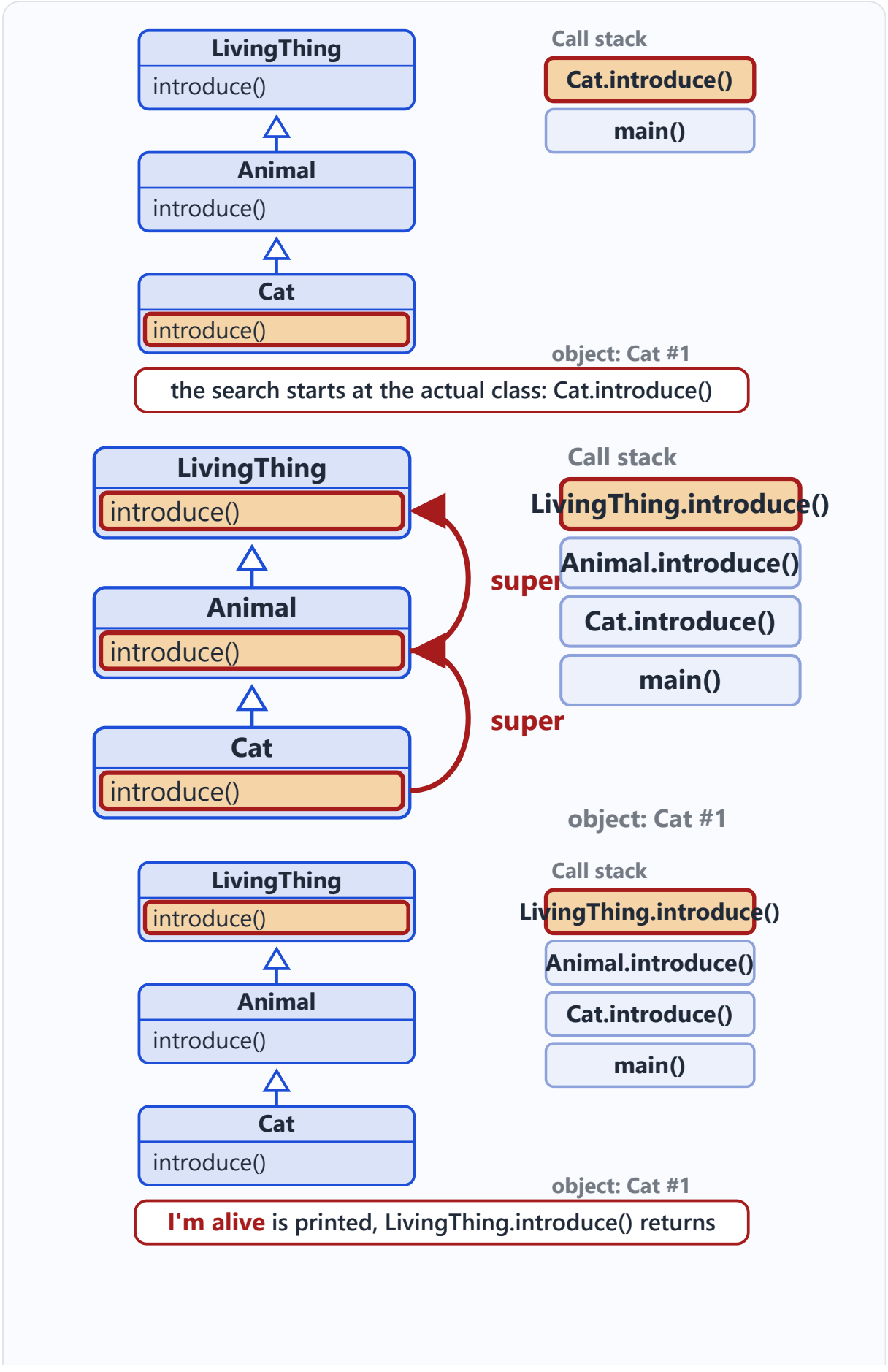
```
class LivingThing {
    void introduce() { System.out.println("I'm alive"); }
}
class Animal extends LivingThing {
    void introduce() {
        super.introduce();
        System.out.println("I'm an animal");
    }
}
class Cat extends Animal {
    void introduce() {
        super.introduce();
        System.out.println("I'm a cat");
    }
}
new Cat().introduce();   // main
```

```
I'm alive
I'm an animal
I'm a cat
```

At first glance the order of the output looks reversed. The call starts at `Cat`, yet the first line printed is `LivingThing`'s. Table 4.3.2 traces the calls step by step.

Step	Event	Call stack	Output
1	The object is a `Cat`; the search starts at the bottom, and `Cat.introduce()` is found.	main, Cat.introduce	—
2	`super.introduce()`: `Animal.introduce()` is called. `Cat.introduce` waits.	main, Cat.introduce, Animal.introduce	—
3	The `super` inside `Animal` points fixedly to `LivingThing`: `LivingThing.introduce()`.	main, Cat.introduce, Animal.introduce, LivingThing.introduce	—
4	The topmost version prints and returns.	main, Cat.introduce, Animal.introduce	I'm alive
5	`Animal.introduce()` resumes where it left off, prints and returns.	main, Cat.introduce	I'm an animal
6	`Cat.introduce()` prints its own line and returns.	main	I'm a cat

Table 4.3.2: Trace of the call `new Cat().introduce()`. In the stack column the topmost frame is on the right.


LivingThing
introduce()
Animal
introduce()
Cat
introduce()
Call stack
Cat.introduce()
main()
object: Cat #1
the search starts at the actual class: Cat.introduce()
LivingThing
introduce()
Animal
introduce()
Cat
introduce()
super
super
Call stack
LivingThing.introduce()
Animal.introduce()
Cat.introduce()
main()
object: Cat #1
LivingThing
introduce()
Animal
introduce()
Cat
introduce()
Call stack
LivingThing.introduce()
Animal.introduce()
Cat.introduce()
main()
object: Cat #1
I'm alive is printed, LivingThing.introduce() returns

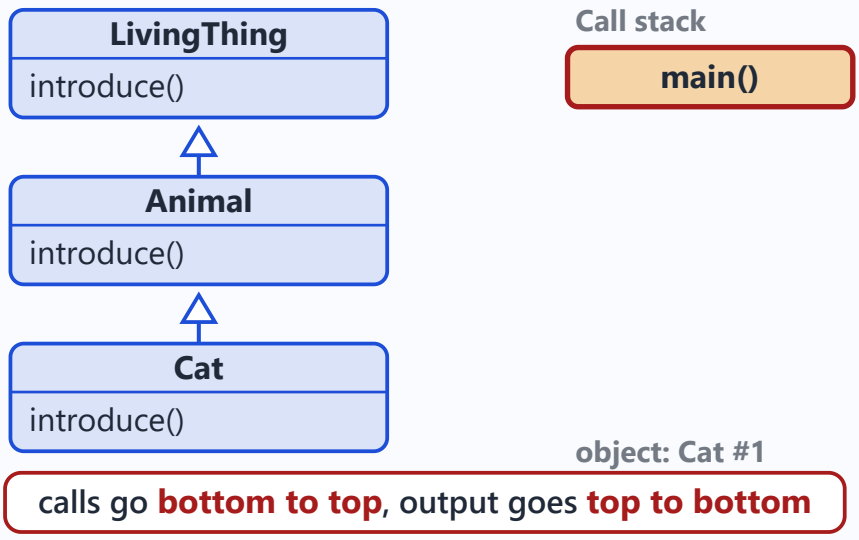


Figure 4.3.8: Four moments of the `super.introduce()` chain. (a) `Cat.introduce()` runs. (b) Two `super` calls push `Animal.introduce` and `LivingThing.introduce` onto the stack. (c) `LivingThing` prints the first line. (d) All the calls have returned; the outputs came in reverse order.

The third step carries a subtle point. The object is still a `Cat`. In an ordinary method call the search starts from the object's actual class. If this rule were followed, the call inside `Animal` would go to `Cat.introduce()` again and an infinite loop would result. `super` does not follow this rule. The `super` written in the code of `Animal` always points to the superclass of `Animal`, that is `LivingThing`, whatever the class of the object is. This link is fixed at compile time; it is not chosen at run time by looking at the object.

Figure 4.3.8 shows the general form of the chain. The calls go in the direction `Cat` → `Animal` → `LivingThing`, and the outputs come in reverse order. Each class first has the superclass do its work, then adds its own difference. The stack reaches at most four frames: `main` and three `introduce` frames. Each level of the chain adds one call and one frame. So for a chain of d levels, the running time and the stack use are proportional to d, that is O(d). Had `new Animal().introduce()` been called, the chain would be one level shorter and only “I'm alive” and “I'm an animal” would be printed.

4.3.3.3 Code trace: the constructor chain

Constructors form a similar chain. In the code below, the constructor of `Animal` writes `super()` explicitly. Had it not, the compiler would have inserted the same line itself.

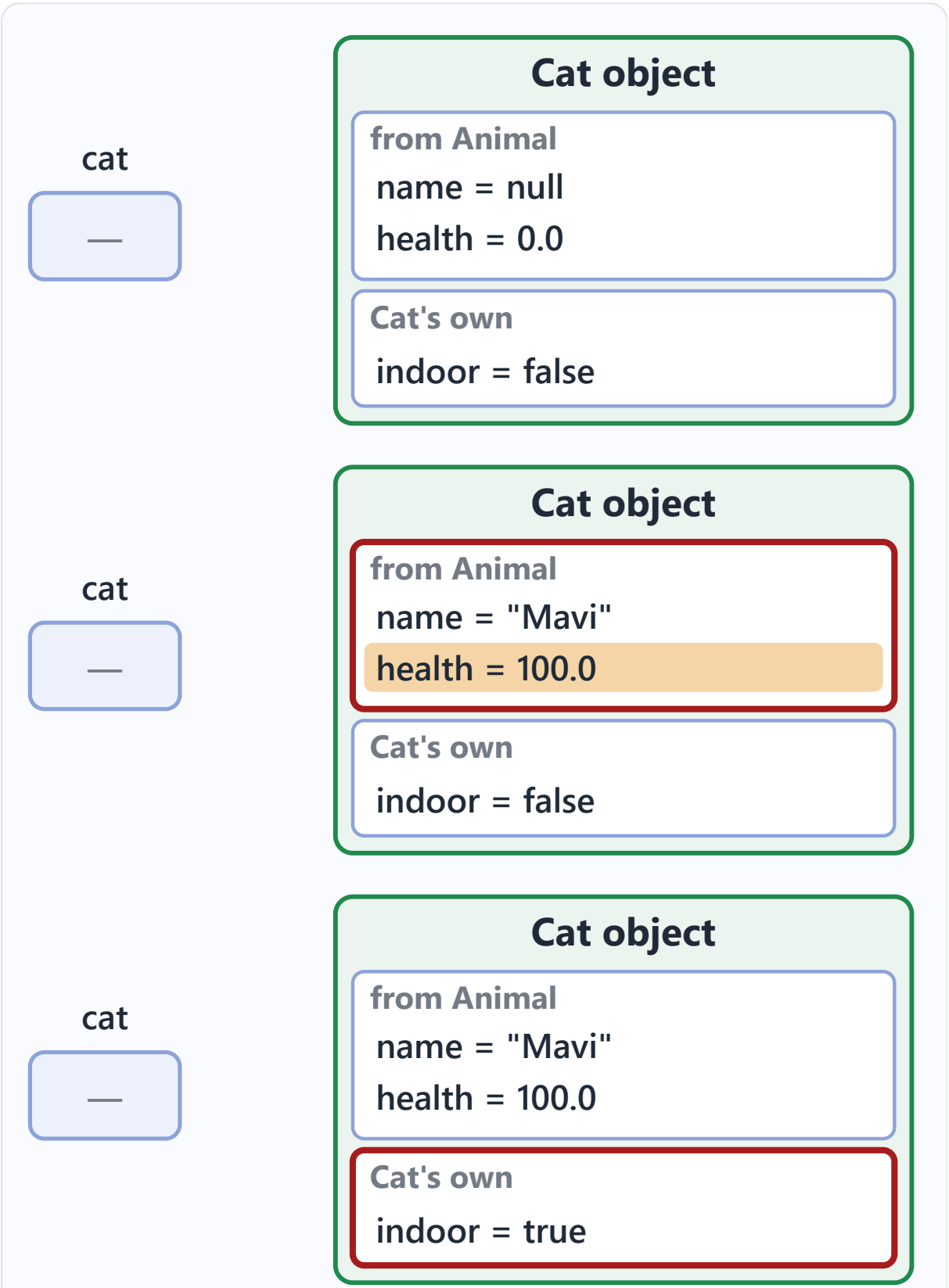
```
class Animal {
    private String name;
    private double health;
    Animal(String name) {
        super();   // Object() is called
        this.name = name;
        health = 100.0;
        System.out.println("Animal constructor");
    }
}
class Cat extends Animal {
    private boolean indoor;
    Cat(String name, boolean indoor) {
        super(name);
        this.indoor = indoor;
        System.out.println("Cat constructor");
    }
}
// in main
Cat cat = new Cat("Mavi", true);
System.out.println("Object ready");
```

```
Animal constructor
Cat constructor
Object ready
```

The evaluation of the expression `new Cat("Mavi", true)` can be examined in eleven steps. Table 4.3.3 gives these steps together with the current values of the object's fields.

Step	Code running	name	health	indoor	Output
1	`new`: space is reserved for the object	null	0.0	false	
2	`Cat(...)` is called	null	0.0	false	
3	`super(name)`: `Animal(...)`; `Cat` waits	null	0.0	false	
4	`super()`: `Object()`	null	0.0	false	
5	`Object()` returned; `this.name = name`	"Mavi"	0.0	false	
6	`health = 100.0`	"Mavi"	100.0	false	
7	`Animal` prints and returns	"Mavi"	100.0	false	Animal constructor
8	`Cat` resumes: `this.indoor = indoor`	"Mavi"	100.0	true	
9	`Cat` prints	"Mavi"	100.0	true	Cat constructor
10	The object's address is assigned to `cat`	"Mavi"	100.0	true	
11	`main` resumes	"Mavi"	100.0	true	Object ready

Table 4.3.3: Trace of `new Cat("Mavi", true)`. The fields are, in order, `name`, `health` and `indoor`.


cat
—
Cat object
from Animal
name = null
health = 0.0
Cat's own
indoor = false
cat
—
Cat object
from Animal
name = "Mavi"
health = 100.0
Cat's own
indoor = false
cat
—
Cat object
from Animal
name = "Mavi"
health = 100.0
Cat's own
indoor = true

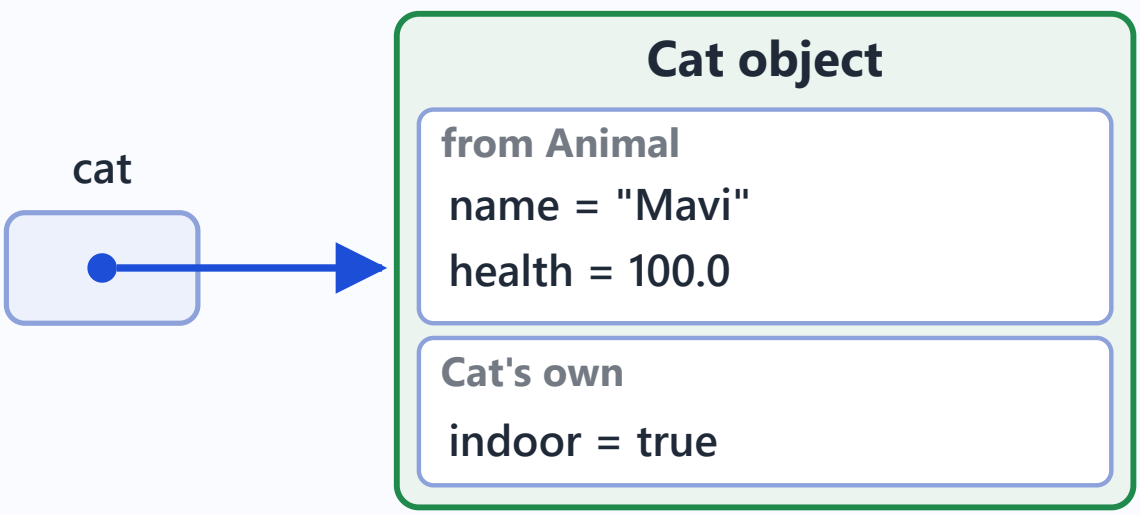


Figure 4.3.9: Four moments of the constructor chain. (a) `new` reserved space for the object; all fields hold default values. (b) `Animal` filled in its own fields. (c) `Cat` filled in its own field. (d) The address of the object was assigned to the variable `cat`.

The first step of the trace is important. `new` reserves space for all the fields of the object *before* any constructor runs. The `name` and `health` that come from `Animal` are included. All fields take their default values: references `null`, numbers 0, `boolean` `false`. The first frame of Figure 4.3.9 shows this state. Constructors do not create fields; they only write values into existing fields.

Then the calls go from the bottom up: `Cat` → `Animal` → `Object`. Since each constructor first calls the constructor of its superclass, none of them can get to its body and all wait. The chain always ends at `Object`. `Object()` has no superclass and its body is empty; it returns at once. After that the bodies complete from the top down: first `Animal` fills in its own fields and prints its message, then `Cat`. The result can be said in one sentence: constructors **complete from the top down**, `Object` → `Animal` → `Cat`.

There is a reason for this order. The constructor of a subclass must be able to use the fields and methods that come from the superclass. For example, the constructor of `Cat` may want to call `getName()` and print a message. For that to be safe, the `Animal` part must already be built. Completing the superclass first guarantees this.

In the last step the object is ready and its address is assigned to the variable `cat`. While the constructor is running, the variable `cat` holds no value yet. A half-built object never reaches the variable in `main`.

4.3.3.4 The implicit `super()` and its errors

The `super();` call that the compiler inserts quietly is often useful, but it is also the source of the compile error beginners meet most often. If the superclass has no no-argument constructor, the implicit call has nothing to match:

```
class Animal {
    private String name;
    Animal(String name) { this.name = name; }
}
class Cat extends Animal {
    Cat(String name) {
        // the compiler inserts super(); here
        System.out.println("Cat");
    }
}
```

```
error: constructor Animal in class Animal
  cannot be applied to given types;
  required: String
  found:    no arguments
```

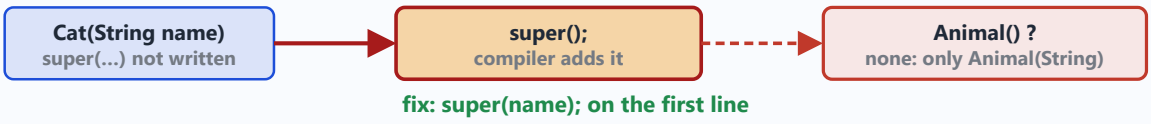


Figure 4.3.10: Because `super(...)` is not written in the constructor of `Cat`, the compiler inserts `super();`. Since `Animal` has only `Animal(String)`, there is no constructor for this call to go to.

The message is confusing at first. The programmer did not call the constructor of `Animal`, yet the compiler says a call was made with the wrong arguments. What the message says is this: the constructor of `Animal` expects a `String` (`required: String`), but the call has no arguments at all (`found: no arguments`). The call without arguments is the `super();` that the compiler inserted, as seen in Figure 4.3.10.

The same error appears when no constructor is written in the subclass at all:

```
class Animal {
    private String name;
    Animal(String name) { this.name = name; }
}
class Cat extends Animal { }
```

```
Main.java:5: error: constructor Animal in class Animal cannot be applied to given types;
class Cat extends Animal { }
^
  required: String
  found:    no arguments
```

For a class with no constructor written at all, the compiler adds a no-argument **default constructor**. The body of this constructor is just `super();`. So the constructor added for `Cat` also looks for the no-argument `Animal()` and cannot find it. The default constructor is added only if the class has *no* constructor at all. The moment `Animal(String)` is written in `Animal`, its no-argument constructor is gone.

The second kind of error is putting `super(...)` in the wrong place:

```
Cat(String name) {
    System.out.println("Cat");
    super(name);
}
```

```
error: call to super must be first
  statement in constructor
```

In Java 21, `super(...)` must be the first statement of the constructor. Not even a `println` can be written before it. The fix is the same for both errors: on the first line of the subclass constructor, call a suitable superclass constructor explicitly, that is, write `super(name);`. Another fix is to add a no-argument constructor to the superclass. But that route means letting `Animal` create a nameless animal, which is usually not wanted.

Common mistake

A programmer who sees the message `constructor Animal in class Animal cannot be applied to given types` usually looks at the `new` line that creates the `Cat` and searches for the error there. The error is in the constructor of class `Cat`: the parameterized constructor of the superclass was not called. The line number in the message points to this; the line is the constructor declaration of `Cat`.

The change that came with Java 25 removes this second error. In Java 25, `System.out.println("Cat")` may be written before `super(name)`; this statement does not touch the object under construction. But even in such statements `this` cannot be used, methods of the object cannot be called and its fields cannot be read. The core of the rule does not change: the object is not used before its superclass part is built. The first error is fixed by no version; the parameterized constructor of the superclass must still be called explicitly.

4.3.3.5 Code trace: `this(...)` and `super(...)`

A constructor can call another constructor of the same class with `this(...)`. This is also a form of **constructor chaining**, and it gathers repeated initialization code in one constructor. Like `super(...)`, `this(...)` must be the first statement. So a constructor starts with either `this(...)` or `super(...)`, not both. In the code below, both classes have two constructors:

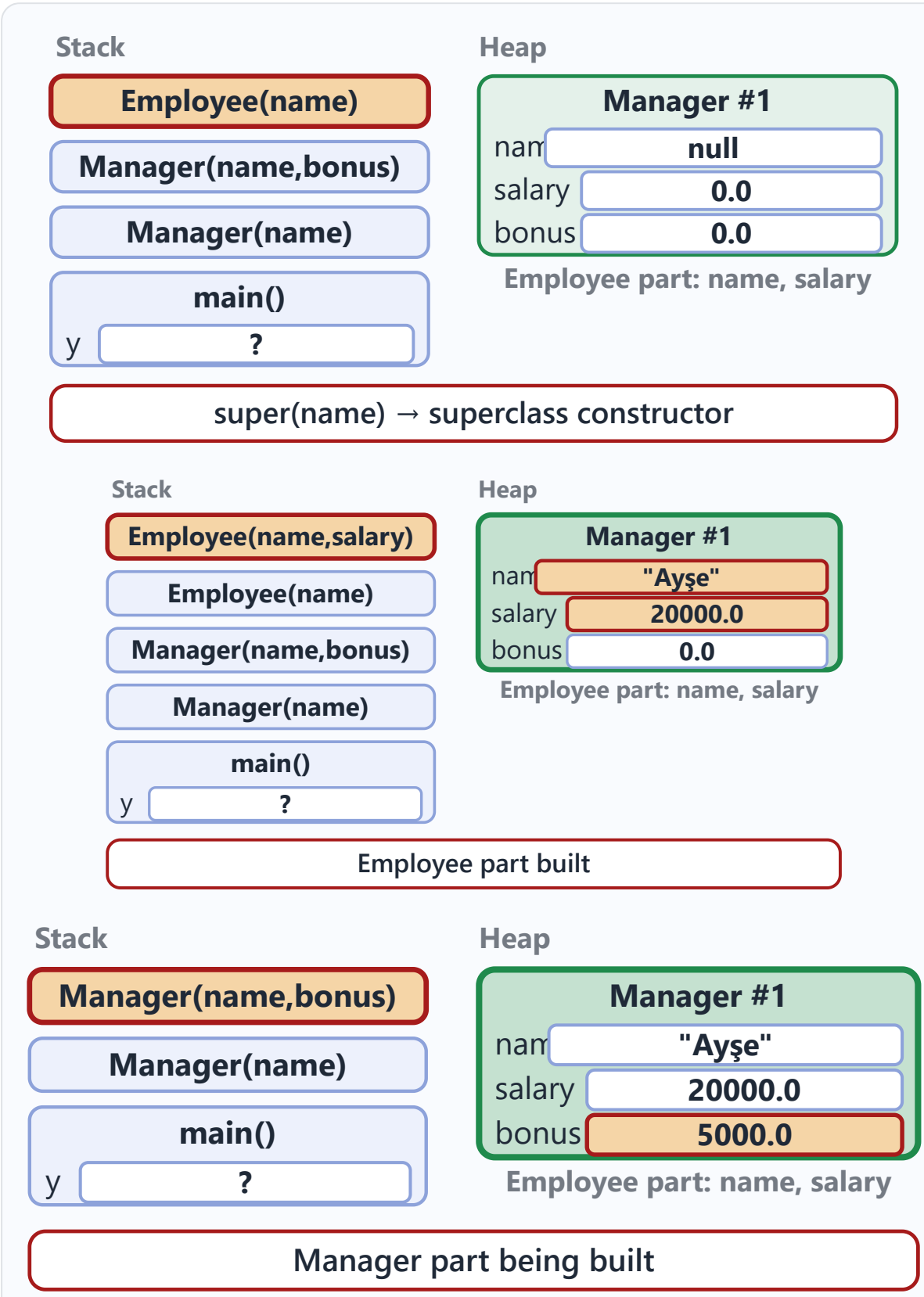
```
class Employee {
    private String name;  private double salary;
    Employee(String name) {
        this(name, 20000.0);
        System.out.println("Employee(String)");
    }
    Employee(String name, double salary) {
        this.name = name;  this.salary = salary;
        System.out.println("Employee(String, double)");
    }
    double getSalary() { return salary; }
}
class Manager extends Employee {
    private double bonus;
    Manager(String name) {
        this(name, 5000.0);
        System.out.println("Manager(String)");
    }
    Manager(String name, double bonus) {
        super(name);
        this.bonus = bonus;
        System.out.println("Manager(String, double)");
    }
    double income() { return getSalary() + bonus; }
}
Manager y = new Manager("Ayşe");   // main
System.out.println(y.income());
```

```
Employee(String, double)
Employee(String)
Manager(String, double)
Manager(String)
25000.0
```

A single `new` runs four constructors. The calls go in this order: `Manager(String)` → `Manager(String, double)` through `this(name, 5000.0)` → `Employee(String)` through `super(name)` → `Employee(String, double)` through `this(name, 20000.0)` → `Object()` through the implicit `super()`. The last constructor in the chain is `Employee(String, double)`; since its first line is neither `this` nor `super`, the compiler inserts `super()` there.

Step	Event	name	salary	bonus	Output
1	`new` ; `Manager(String)` is called	null	0.0	0.0	
2	`this(name, 5000.0)`	null	0.0	0.0	
3	`super(name)` : `Employee(String)`	null	0.0	0.0	
4	`this(name, 20000.0)` , implicit `super()` : `Object()` returns	null	0.0	0.0	
5	Body of `Employee(String, double)`	"Ayşe"	20000.0	0.0	Employee(String, double)
6	Rest of the body of `Employee(String)`	"Ayşe"	20000.0	0.0	Employee(String)
7	`Manager(String, double)` : `bonus = 5000.0`	"Ayşe"	20000.0	5000.0	Manager(String, double)
8	`Manager(String)` finishes	"Ayşe"	20000.0	5000.0	Manager(String)
9	`y.income()` = 20000.0 + 5000.0	"Ayşe"	20000.0	5000.0	25000.0

Table 4.3.4: Trace of `new Manager("Ayşe")` . The fields are `name` , `salary` and `bonus` .


Stack
Heap
Employee(name)
Manager(name,bonus)
Manager(name)
main()
y
?
Manager #1
nam
null
salary
0.0
bonus
0.0
Employee part: name, salary
super(name) → superclass constructor
Stack
Heap
Employee(name,salary)
Employee(name)
Manager(name,bonus)
Manager(name)
main()
y
?
Manager #1
nam
"Ayşe"
salary
20000.0
bonus
0.0
Employee part: name, salary
Employee part built
Stack
Heap
Manager(name,bonus)
Manager(name)
main()
y
?
Manager #1
nam
"Ayşe"
salary
20000.0
bonus
5000.0
Employee part: name, salary
Manager part being built

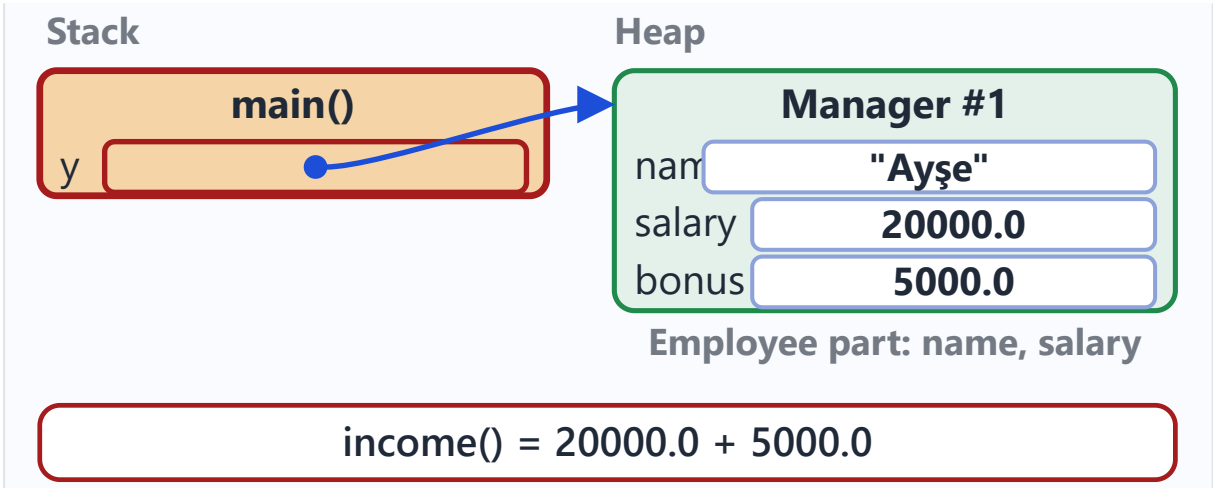


Figure 4.3.11: Four moments of the `this(...)` and `super(...)` trace. (a) `super(name)` : the `Manager` part waits until the `Employee` part is done. (b) The first body runs at the top; `name` and `salary` get filled. (c) `super(name)` has completed; now `bonus` gets filled. (d) The address of the object is assigned to `y` .

Table 4.3.4 and Figure 4.3.11 show the same rule once more. The calls go from the bottom up and the bodies complete starting from the top. Four constructors ran, but at no moment, however many constructors were on the stack, was more than one object built. The `bonus` field stays at 0.0 until step seven. A subclass's own field is not filled until the superclass part is completely finished.

The method `income()` on the last line adds `getSalary()` , which arrives by inheritance, and its own `bonus` field. Since `salary` is `private` , `Manager` cannot read it by name; it uses the getter.

The constructor chain has one limit: it cannot form a cycle. If `Employee(String)` called `this(name, 1.0)` and that called `this(name)` , the chain would never reach `Object` . The compiler catches this:

```
class Employee {
    Employee(String name) { this(name, 1.0); }
    Employee(String name, double m) { this(name); }
}
```

```
error: recursive constructor invocation
```

Writing both `this(...)` and `super(...)` in the same constructor is forbidden too. Each wants to be the first statement:

```
Manager(String name) {
    this(name, 5000.0);
    super(name);
}
```

```
error: call to super must be first statement in constructor
```

Of the constructors in a chain, only the last one calls `super(...)`. The others delegate to it with `this(...)`.

4.3.3.6 The order of initialization in inheritance

Constructors are not the only code that runs when an object is built. The initial values in field declarations (`int a = value(...)`) and `static` blocks run too. Once inheritance is involved, the order of these pieces matters. The code below makes the order visible. Each piece prints a trace when it runs:

```
class Parent {
    static { System.out.println("Parent static"); }
    int a = value("Parent field");
    Parent() { System.out.print("Parent(), "); }
    static int value(String s) {
        System.out.print(s + ", ");
        return 1;
    }
}
class Child extends Parent {
    static { System.out.println("Child static"); }
    int b = value("Child field");
    Child() { System.out.println("Child()"); }
}
// in main:
new Child();
new Child();
```

```
Parent static
Child static
Parent field, Parent(), Child field, Child()
Parent field, Parent(), Child field, Child()
```

The output separates two phases: class initialization and object construction.

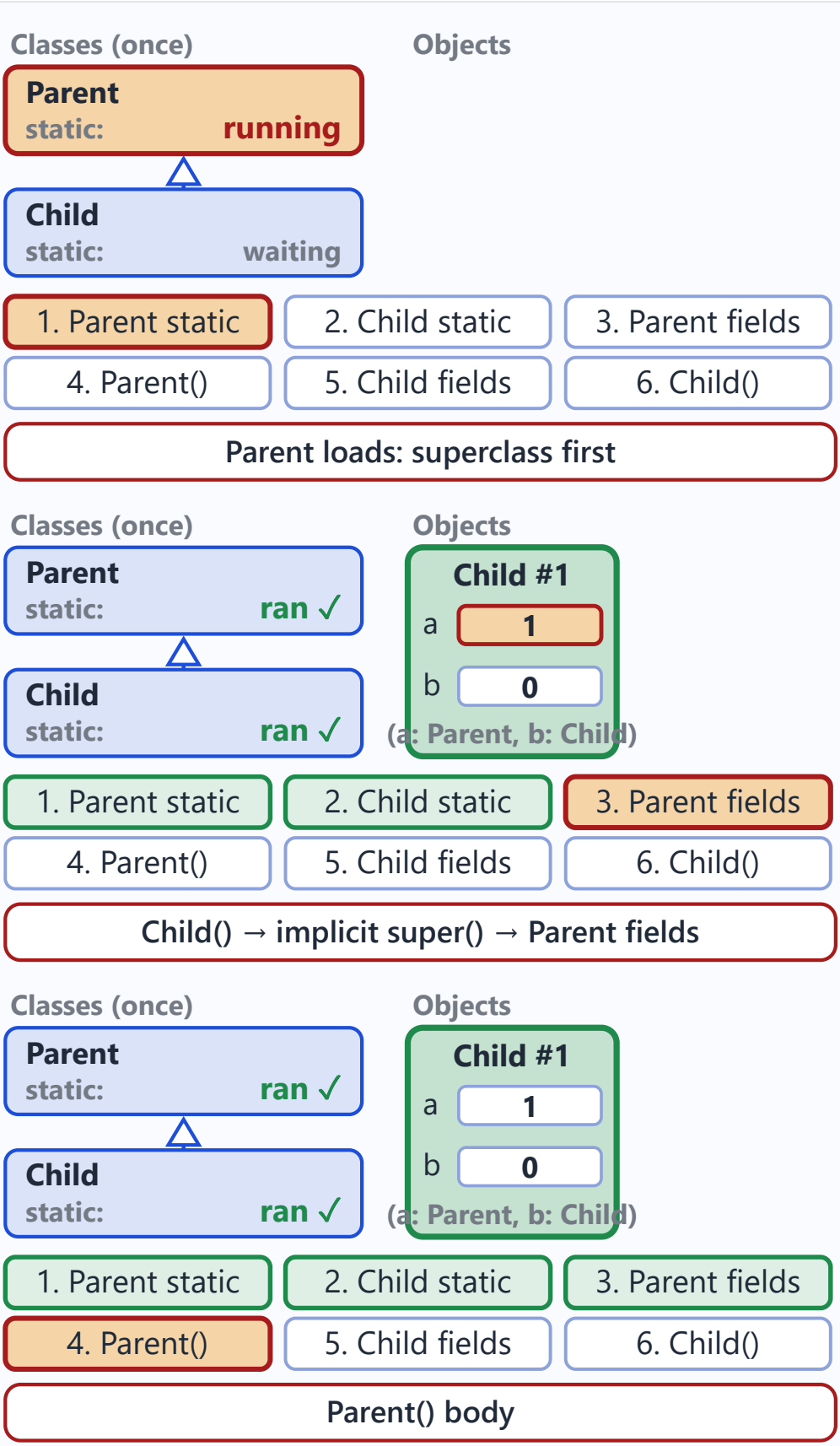
Class initialization. Before the JVM (Java Virtual Machine) uses a class for the first time, it initializes it. At this point the `static` blocks of the class and the initial values of its `static` fields run. Before a subclass is initialized, its superclass is initialized. That is why “Parent static” is printed first and “Child static” after it. Class initialization happens **once** per program run.

Object construction. For every `new`, these four steps run in this order:

1. the field initializers of the superclass (`a = value("Parent field")`),

2. the constructor body of the superclass (`Parent()`),
3. the field initializers of the subclass (`b = value("Child field")`),
4. the constructor body of the subclass (`Child()`).

The reason is where the compiler puts the field initializers. The compiler places the assignments from field declarations inside every constructor, right after the `super(...)` call. So `Child()` is really compiled like this: first `super()`, then `b = value("Child field")`, then the body. `Parent()` likewise: first `super()`, that is `Object()`, then `a = ...`, then the body.


Classes (once)
Objects
Parent
static: running
Child
static: waiting
1. Parent static
2. Child static
3. Parent fields
4. Parent()
5. Child fields
6. Child()
Parent loads: superclass first
Classes (once)
Objects
Parent
static: ran ✓
Child
static: ran ✓
Child #1
a 1
b 0
(a: Parent, b: Child)
1. Parent static
2. Child static
3. Parent fields
4. Parent()
5. Child fields
6. Child()
Child() → implicit super() → Parent fields
Classes (once)
Objects
Parent
static: ran ✓
Child
static: ran ✓
Child #1
a 1
b 0
(a: Parent, b: Child)
1. Parent static
2. Child static
3. Parent fields
4. Parent()
5. Child fields
6. Child()
Parent() body

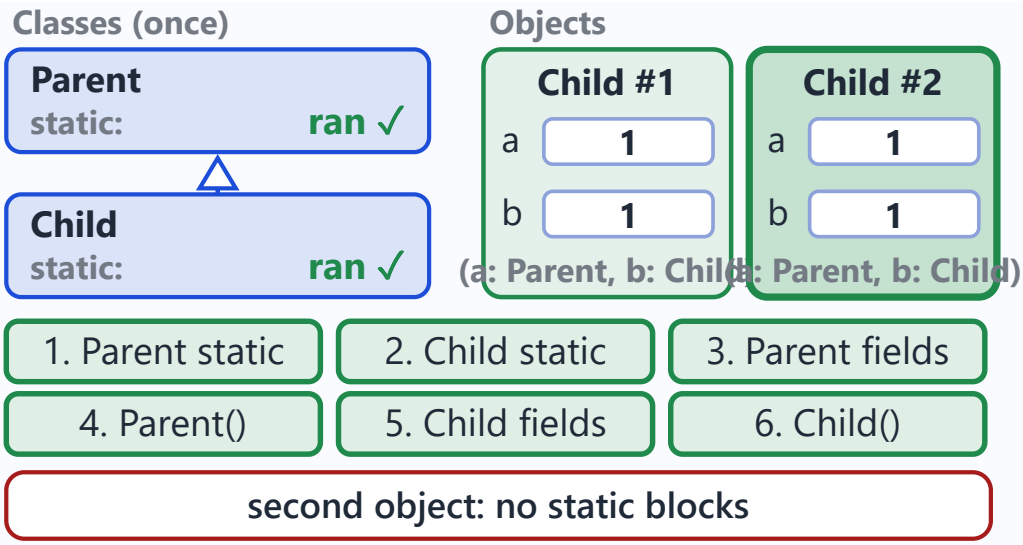


Figure 4.3.12: Four moments of the initialization order. (a) For the first `new Child()`, the `static` block of `Parent` runs first. (b) The object has been allocated; the field `a` of `Parent` got its initial value. (c) While the body of `Parent()` runs, `b` is still 0. (d) The second `new Child()`: the `static` blocks do not run again.

The third frame of Figure 4.3.12 shows a dangerous consequence of this order. While the body of `Parent()` runs, the subclass's field `b` is still 0. The subclass part has not received any initial value yet. This detail matters when the constructor of the superclass calls a method that the subclass overrides:

```
class Parent {
    Parent() { print(); }
    void print() { System.out.println("Parent.print"); }
}
class Child extends Parent {
    int b = 7;
    String s = "ready";
    @Override void print() {
        System.out.println("Child.print b=" + b + " s=" + s);
    }
}
// in main:
Child a = new Child();
a.print();
```

```
Child.print b=0 s=null
Child.print b=7 s=ready
```

`new Child()` runs `Parent()` first. The `print()` call inside `Parent()` goes to `Child.print()`, because the object is a `Child`. But at that moment the field initializers of `Child` have not been assigned yet: `b` is 0 and `s` is `null`. Although the source says `int b = 7`, the method sees 0. The

second call, made after the object is built, sees the right values.

Common mistake

Calling an overridable method in the constructor of a superclass. If a subclass overrides this method, the method runs before the subclass's fields have received their initial values. The result is a wrong computation on default values, or a `NullPointerException`. If the methods called from a constructor are made `private`, `final` or `static`, they cannot be overridden and the problem goes away.

Example 4.3.3

Instance initializer blocks (`{ ... }`) run in the same place as field initializers, right after `super(...)`. Given these classes:

```
class A {
    static { System.out.print("As "); }
    { System.out.print("Ab "); }
    A() { System.out.print("A "); }
}
class B extends A {
    static { System.out.print("Bs "); }
    { System.out.print("Bb "); }
    B() { this(0); System.out.print("B "); }
    B(int x) { System.out.print("B" + x + " "); }
}
```

What does the first `new B()` in the program print? First the classes are initialized: `As Bs`. Then `B()` is called. Its first line is `this(0)`, so it goes to `B(int)` without entering its own body. On the first line of `B(int)` there is an implicit `super()`: the instance block and body of `A` run, and `Ab A` is printed. After `super()` returns, the instance block of `B` runs: `Bb`. Then the body of `B(int)` prints `B0`, and last the body of `B()` prints `B`. Result: `As Bs Ab A Bb B0 B`.

The instance block runs not in `B()` but only in the constructor that calls `super(...)`, that is `B(int)`. So `Bb` is printed once. If `new B(7)` is called afterwards, the `static` blocks do not run again: `Ab A Bb B7`.

4.3.4 Overriding

A subclass does not always want to use a behavior it took from the superclass as it is. A dog does not say “This animal speaks.”; it barks. A rectangle is not drawn like a generic shape but as a rectangle. This section explains how a subclass rewrites a method it received by inheritance, the rules this is subject to, and why those rules exist.

4.3.4.1 Overriding a method

Definition 4.3.3

Overriding is the subclass giving its own behavior to a method it inherited from the superclass by rewriting that method **with the same signature**. The signature of a method is its name and parameter list (the number, types and order of the parameters).

Overriding has four basic conditions:

- The name and parameter list must be the same.
- The return type must be the same or a subtype of it.
- Access cannot be narrowed: a `public` method must stay `public` in the subclass.
- `final`, `static` and `private` methods cannot be overridden.

```
class Shape {
    public void draw() {
        System.out.println("Drawing a shape.");
    }
}
class Rectangle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle.");
    }
}
// in main
new Shape().draw();
new Rectangle().draw();
```

```
Drawing a shape.
Drawing a rectangle.
```

`Rectangle` rewrites `draw()` with the same name, the same (empty) parameter list, the same return type and the same access. `new Shape().draw()` runs the superclass version, `new Rectangle().draw()` the subclass version. The actual class of the object decides which version runs. Even when the variable's type is `Shape`, nothing changes: `Shape s = new Rectangle(); s.draw();` still prints “Drawing a rectangle.” This behavior is called **dynamic binding** and is the foundation of polymorphism.

If one of the conditions is broken, the compiler reports an error. Forgetting to write `public` in the subclass narrows the access to package level:

```
// in subclass: void draw()
error: draw() in Rectangle cannot override
  draw() in Shape
  attempting to assign weaker access
  privileges; was public
```

Changing the return type breaks overriding too:

```
// in subclass: public int draw()
error: draw() in Rectangle cannot override
  draw() in Shape
  return type int is not compatible
  with void
```

The first line of both messages is the same: `draw() in Rectangle cannot override draw() in Shape`. The compiler saw two methods with the same signature and counted this as an attempt to override. The second line says which rule was broken. When reading the message, the real information is on this second line.

4.3.4.2 Overriding and overloading

Overriding is often confused with another concept with a similar name: **overloading**. In overloading, several methods with the same name but different parameter lists are written. The difference between the two is summarized in Table 4.3.5.

	Overriding	Overloading
Where?	In the subclass, for an inherited method	In the same class (or a subclass)
Name	Same	Same
Parameters	Same	Different (number or type)
Return type	Same or a subtype	Free
Purpose	Change an inherited behavior	Do the same job with different inputs

	Overriding	Overloading
When chosen	Run time (actual type of the object)	Compile time (types of the arguments)

Table 4.3.5: Comparison of overriding and overloading.

Both can also be present in the same class:

```
class Rectangle extends Shape {
    @Override
    public void draw() { ... }             // overriding
    public void draw(String color) { ... }  // overloading
}
```

The first `draw()` overrides the method in `Shape`. The second is `draw(String color)`; because its parameter list is different, it is a new method and overloads `draw()`. In the call `d.draw("blue")` the compiler looks at the argument list: a `String` is given, so `draw(String)` is chosen. The body of this method adds the color to the start of the message:

```
Drawing a blue rectangle.
```

The last row of the table states the most important difference. In overloading, the compiler decides which method to call by looking at the *compile-time* types of the arguments. In overriding, the decision is made while the program runs, by looking at the actual class of the object. Both concepts are a foundation of polymorphism. The first is called compile-time polymorphism and the second run-time polymorphism.

4.3.4.3 The `@Override` annotation

Definition 4.3.4

`@Override` is an **annotation**. It declares that the method overrides a method inherited from a superclass. The compiler checks this declaration: if no method with the same signature exists in the superclasses or interfaces, it reports a compile error.

```
public class Rectangle extends Shape {
    // override the inherited method
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle.");
    }
}
```

The annotation is not mandatory; overriding works even if it is not written. But writing it is always recommended. It has two benefits. First, someone reading the code sees at once that the method comes from the superclass. Second, and more important, it catches typos at compile time. Suppose the name of the method is misspelled:

```
    @Override
    public void Draw() { ... }
```

```
error: method does not override or implement
  a method from a supertype
```

`Draw` starts with a capital letter. Since Java distinguishes upper and lower case, `Draw()` and `draw()` are different methods. What would happen without `@Override`? The compiler would not complain; `Draw()` would silently become a new method. The program would run, but behave wrongly:

```
class Rectangle extends Shape {
    public void Draw() {          // no @Override
        System.out.println("Drawing a rectangle.");
    }
}
// in main:
Shape s = new Rectangle();
s.draw();
```

```
Drawing a shape.
```

For the call `s.draw()`, `Rectangle` has no overridden `draw()`; the search climbs to `Shape` and the generic version runs. A bug of this kind is noticed only when the program is tried out, perhaps much later. With `@Override` written, it would have been caught at compile time. The same happens when a parameter type is written wrongly, for example `equals(Point)` instead of `equals(Object)`.

4.3.4.4 The rules of overriding

The rules of overriding can be memorized one by one, but they all derive from a single reason. Table 4.3.6 gives the rules together with the messages javac gives when they are broken. The messages were obtained with JDK 21.

Property	Allowed in the subclass method	javac if the rule is broken
Name and parameters	Exactly the same	No error: it becomes a new method (overloading)

Property	Allowed in the subclass method	javac if the rule is broken
Return type	The same or a subtype (covariant)	`return type String t not compatible with Animal`
Access	The same or wider: `protected` → `public`	`attempting to assign weaker access privileges`
Checked exception	The same, a more specific one, or none	`overridden method does not throw Exception`
`static`	An instance method cannot be `static`; a `static` method cannot be an instance method	`overriding method t static` / `overridden method t static`
`final` method	Cannot be overridden	`overridden method t final`
`private` method	Invisible; a method with the same name is an unrelated new method	No error (error if `@Override` is present)

Table 4.3.6: The rules of overriding and the javac messages produced when a rule is broken.

The common reason is this: a subclass object must be usable wherever a superclass object is expected. Code that works with a variable of type `Shape` does not know whether the object the variable refers to is really a `Rectangle` or a `Circle`. This code relies only on the promise `Shape` makes. The promise of the superclass has three parts: what the method returns, to whom it is open, and which exceptions it may throw. The subclass may keep this promise or widen it, but cannot narrow it.

With this reason the rules explain themselves one by one. If the superclass's method is `public`, it can be called from anywhere. Had the subclass lowered it to package access, a call from another package through a `Shape` variable would be forbidden when it reached a `Rectangle` object. Widening access creates no problem. Making a `protected` method `public` breaks none of the old callers.

Exceptions follow the same logic. If the superclass method `f()` declares no checked exception, the code that calls it has not written `try`-`catch`. If the subclass version says `throws Exception`, this code would be caught unprepared. The compiler prevents this:

```
class Parent { void f() { } }
class Child extends Parent {
    @Override void f() throws Exception { }
}
```

```
error: f() in Child cannot override f() in Parent
  overridden method does not throw Exception
```

The reverse is allowed. If the superclass declares `throws Exception`, the subclass may declare a more specific exception (`throws IOException`) or none at all. The caller is already ready to catch `Exception`; a more specific exception falls within it. `RuntimeException` and its subclasses are unchecked exceptions; this rule does not apply to them.

Example 4.3.4

Given `class U { protected void a() { } public void b() { } }`. Inside `class V extends U` we write `@Override public void a() { }` and `@Override void b() { }`. Which one compiles?

`a()` is `protected` in the superclass; making it `public` in the subclass widens the access, so it compiles. `b()` is `public` in the superclass; writing it without a modifier lowers the access to package level. javac gives the messages `b() in V cannot override b() in U` and `attempting to assign weaker access privileges; was public`. The order of access from narrow to wide is: `private`, package, `protected`, `public`.

4.3.4.5 Covariant return type

The return type rule looks strict on first reading, but it has some flexibility. An overriding method may return a **subtype** of the return type in the superclass:

```
class Animal {
    Animal reproduce() { return new Animal(); }
    public String toString() { return "Animal"; }
}
class Cat extends Animal {
    @Override
    Cat reproduce() { return new Cat(); }  // subtype
    void meow() { System.out.println("meow"); }
    public String toString() { return "Cat"; }
}
// in main:
Cat k = new Cat();
k.reproduce().meow();   // no cast needed
Animal h = k;
System.out.println(h.reproduce());
```

```
meow
Cat
```

Definition 4.3.5

A **covariant return type** means that an overriding method returns a subtype of the return type declared in the superclass.

`Animal.reproduce()` returns an `Animal`. `Cat.reproduce()` returns a `Cat`; since `Cat` is a subtype of `Animal`, this is a valid override. The benefit shows at the first call. A caller that uses a `Cat` reference receives the result directly as a `Cat`. `k.reproduce().meow()` can be written, with no `(Cat)` cast. If the return type had stayed `Animal`, the call to `meow()` would not compile, because `Animal` has no such method.

For a caller that uses an `Animal` reference, the promise is not broken. The compile-time type of `h.reproduce()` is `Animal`, and what comes back really is an animal, because a `Cat` is an `Animal` too. Since the object is a `Cat`, the call goes to `Cat.reproduce()`; `println` calls the `toString()` of the new cat and prints "Cat".

Covariance applies only to reference types. A method that returns `int` cannot be overridden to return `long`; there is no subtype relation between primitive types. An unrelated reference type is not allowed either:

```
error: reproduce() in Cat cannot override
  reproduce() in Animal
  return type String is not compatible with Animal
```

`String` is not a subtype of `Animal`. Code that calls `reproduce()` through an `Animal` variable expects an animal and would get a string. The compiler does not allow this.

4.3.4.6 `private` and `static` methods

A `private` method is invisible to the subclass. A method written in the subclass with the same name does not override it; it is a new method, unrelated to it. This sometimes produces unexpected results:

```
class Parent {
    private void greet() { System.out.println("Parent.greet"); }
    void call() { greet(); }
}
class Child extends Parent {
    void greet() { System.out.println("Child.greet"); } // new
}
// in main:
Child a = new Child();
a.call();
a.greet();
```

```
Parent.greet
Child.greet
```

`a.greet()` calls the method of `Child`, as expected. The surprise is the result of `a.call()`. `call()` is written in `Parent`, and `Child` inherits it. The call to `greet()` inside it is bound at compile time to `Parent`'s own `private` method. Since `private` methods cannot be overridden, the compiler does not leave this call to run time. Even though the object is a `Child`, "Parent.greet" is printed.

If `greet()` in `Parent` were not `private`, the situation would reverse:

```
class Parent {
    void greet() { System.out.println("Parent.greet"); }
    void call() { greet(); }
}
class Child extends Parent {
    @Override void greet() { System.out.println("Child.greet"); }
}
// in main:
new Child().call();
```

```
Child.greet
```

Here `Child.greet()` really does override. The call in `call()` is chosen by the actual class of the object, and the version in `Child` runs. The only difference between the two programs is the word `private`, but the behavior changes completely.

If a programmer thinks `Child.greet()` overrides, the `@Override` habit exposes the misunderstanding. If `@Override` is added to `Child.greet()` in the `private` example, the compiler gives the message `method does not override or implement a method from a supertype`. The annotation says "I think I am overriding, please check"; the compiler reports the result of the check.

The same happens with `static` methods. A `static` method with the same signature in the subclass does not override the one in the superclass; it only **hides** it. Which one runs is decided not by the object but by the compile-time type of the variable:

```
class Parent {
    static void f() { System.out.println("Parent.f"); }
    void g() { System.out.println("Parent.g"); }
}
class Child extends Parent {
    static void f() { System.out.println("Child.f"); }
    @Override void g() { System.out.println("Child.g"); }
}
// in main:
Parent u = new Child();
u.f();
u.g();
```

```
Parent.f
Child.g
```

`u` refers to a `Child` object, but its type is `Parent`. The `static` method `f()` belongs to the class. The compiler reads `u.f()` as `Parent.f()` and never looks at the object. The instance method `g()`, on the other hand, is overridden; the actual class of the object makes the choice. That is why calling a `static` method through an object variable is misleading. The right way to write it is `Parent.f()` or `Child.f()`. Writing an instance method and a `static` method with the same signature, in either order, is a compile error:

```
error: f() in Child cannot override f() in Parent
  overriding method is static
error: f() in Child cannot override f() in Parent
  overridden method is static
```

4.3.4.7 Example: a product hierarchy

The power of overriding shows in calls the superclass makes from inside itself. In a store program, let the selling price of products be computed by adding VAT (value-added tax). Food products have a different VAT rate; some products carry a discount.

```
class Product {
    private String name;
    private double price;
    Product(String name, double price) {
        this.name = name; this.price = price;
    }
    double taxRate() { return 0.20; }
    double salePrice() { return price * (1 + taxRate()); }
    @Override public String toString() {
        double f = Math.round(salePrice() * 100) / 100.0;
        return name + ": " + f + " TL";
    }
}
class Food extends Product {
    Food(String name, double price) { super(name, price); }
    @Override double taxRate() { return 0.01; }
}
class SaleProduct extends Product {
    private double discount;
    SaleProduct(String name, double price, double discount) {
        super(name, price);
        this.discount = discount;
    }
    @Override double salePrice() {
        return super.salePrice() * (1 - discount);
    }
    @Override public String toString() {
        return super.toString()
            + " (" + (int) (discount * 100) + " % off)";
    }
}
```

Class `Product` splits the price computation into two steps. `taxRate()` gives the rate, and `salePrice()` computes with that rate. `toString()` rounds the selling price to two decimals and prints it with the name. For rounding, the pattern `Math.round(x * 100) / 100.0` is used. This pattern cleans up the long tails floating-point arithmetic can produce; for example, the result of `0.1 + 0.2` is 0.30000000000000004.

The two subclasses follow two different paths. `Food` changes only `taxRate()`. `SaleProduct` overrides `salePrice()` but does not replace the computation of the superclass; it **extends** it. It applies the discount to the result of `super.salePrice()`. It extends `toString()` too, and appends the discount rate at the end.

```
Product[] cart = {new Product("Pen", 50),
    new Food("Bread", 10),
    new SaleProduct("Headphones", 1000, 0.25)};
for (Product u : cart) System.out.println(u);
```

```
Pen: 60.0 TL
Bread: 10.1 TL
Headphones: 900.0 TL (25% off)
```

Product	`taxRate()`	`salePrice()`	Computation	Result
Pen (`Product`)	`Product` : 0.20	`Product`	50 × 1,20	60.0
Bread (`Food`)	`Food` : 0.01	`Product`	10 × 1,01	10.1
Headphones (`SaleProduct`)	`Product` : 0.20	`SaleProduct` → `super`	1000 × 1,20 × 0,75	900.0

Table 4.3.7: Which method runs for each product in the cart, and the computation.

The second row of Table 4.3.7 is the most instructive part of the example. `salePrice()` and `toString()` are not written in `Food`. Both come from `Product`. But the call to `taxRate()` inside `salePrice()` means `this.taxRate()`, and `this` is a `Food`. The call is chosen by the class of the object; the 1% rate is used. The code of the superclass ends up calling a method written by a subclass. When the superclass was written, `Food` perhaps did not exist yet.

In the third row the chain has two levels. `toString()` is first the one in `SaleProduct`, and it calls the one in `Product` with `super.toString()`. `Product.toString()` calls `salePrice()`; since the object is a `SaleProduct`, `SaleProduct.salePrice()` runs. It has `Product` do its computation with `super.salePrice()`: 1000 × 1.20 = 1200. Then it applies the discount: 1200 × 0.75 = 900. The discount rate is converted to an integer with `(int) (discount * 100)` and "25%" is printed.

Example 4.3.5

Consider three products priced at 200 TL: a plain `Product`, a `Food` and a `SaleProduct` with a 10% discount. What are their selling prices?

`Product`: 200 × 1.20 = 240.0. `Food`: `taxRate()` returns 0.01, so 200 × 1.01 = 202.0. `SaleProduct`: first `super.salePrice()` = 240, then 240 × 0.90 = 216.0. The discount is applied to the price with VAT. Applying it to the price without VAT would not change the result, because the order of multiplication does not affect the product: 200 × 0.90 × 1.20 is also 216.

This design shows that the superclass can be extended with new kinds of products without being closed to change. To add a product with a luxury consumption tax, a new subclass overrides `taxRate()` or `salePrice()`. There is no need to touch `Product` or the loop that walks through the cart.

4.3.5 Access and restriction

Inheritance opens the inside of the superclass to subclasses to some degree. How much it opens is part of the design. This section first describes `protected`, an access level made for subclasses. It then covers `final`, which closes inheritance completely, and narrowing the view of an object through its superclass type.

4.3.5.1 The `protected` access level

Java has four access levels. `public` members are reachable from everywhere. `private` members are reachable only from inside their class. Members with no modifier have **package access**; all classes in the same package see them. `protected` is a level between package access and `public`.

Modifier	Same class	Same package	Subclass (other package)	Everywhere
`private`	✓	✗	✗	✗
(no modifier: package)	✓	✓	✗	✗
`protected`	✓	✓	✓	✗
`public`	✓	✓	✓	✓

Table 4.3.8: Scope of the access modifiers (✓ accessible, ✗ not accessible).

> **Definition 4.3.6**
>
> A `protected` member can be accessed by the class itself, by all classes in the same package, and by subclasses in other packages.

Table 4.3.8 shows a wider scope in each row than in the one before. The word `protected` sounds restrictive, but as the table shows, it is *more open* than package access. This is often confused. `protected` adds one more permission on top of package access: it opens the member to subclasses in other packages.

A subclass can reach the `private` fields of a class only through the `public` or `protected` methods it inherits from the superclass. Good practice is this: the fields stay `private`, and the operations subclasses need are opened as `protected` methods. Making a field `protected` looks easy, but the responsibility for checking the field's value is now spread over all subclasses. A subclass can set health to −50, and `Animal` has no chance to prevent it.

4.3.5.2 `protected`: a code example

The two classes below are in different packages. `Animal` is in package `animals`, `Cat` in package `indoor`.

```java
package animals;

public class Animal {
    protected String name;  // subclasses
    private double health;  // Animal only

    public Animal(String name) {
        this.name = name;
        health = 100.0;
    }
    public double getHealth() { return health; }
    protected void heal(double amount) {
        health += amount;
    }
}
```

```java
package indoor;     // another package
import animals.Animal;

public class Cat extends Animal {
    public Cat(String name) { super(name); }
    public void sleep() {
        // protected field: allowed
        System.out.println(name + " is sleeping.");
        // health += 5;  // private: error
        heal(5);   // protected method
        System.out.println("Health: "
                           + getHealth());
    }
}
```

The two fields of `Animal` show two different design choices. `name` is `protected`; subclasses can read it directly. `health` is `private`. It can be reached in only two ways: the public `getHealth()` reads it, and `heal(double)`, open only to subclasses, changes it.

The `sleep()` method of `Cat` uses both ways. It writes `name` directly; although `Cat` is in another package, it is a subclass of `Animal`. It raises health with `heal(5)` and reads it with `getHealth()`. `new Cat("Mavi").sleep();` prints:

```
Mavi is sleeping.
Health: 105.0
```

If the `health += 5;` in the comment is uncommented, compilation stops:

```
error: health has private access in Animal
```

The benefit of this design is concrete. The owner of `Animal` may later want health never to exceed 100. They write this rule only in `heal`; all subclasses comply automatically. If the field were `protected`, the code of every subclass would have to be checked one by one.

A class that is not a subclass cannot reach a `protected` member from another package:

```
package other;
import animals.Animal;
public class Keeper {              // not a subclass
    String name(Animal h) { return h.name; }
}
```

```
other/Keeper.java:4: error: name has protected access in Animal
```

`Keeper` is neither in package `animals` nor a subclass of `Animal`. None of the four columns of the table allows it. If `Keeper` were moved to package `animals`, it would compile: classes in the same package can reach `protected` members.

4.3.5.3 The fine print of `protected` access

One rule of `protected` does not fit in the table. A subclass in another package can reach a `protected` member only through a reference of **its own type** (or its own subtypes):

```
package animals;
public class Animal {
    protected String name;
    public Animal(String name) {
        this.name = name;
    }
}
```

```
package indoor;
import animals.Animal;
public class Cat extends Animal {
    public Cat(String name) { super(name); }
    void compare(Cat k, Animal h) {
        System.out.println(name);    // its own field
        System.out.println(k.name);  // through Cat
        System.out.println(h.name);  // through Animal
    }
}
```

```
indoor/Cat.java:8: error: name has protected access in Animal
        System.out.println(h.name);
```

The three lines in the method `compare` use three different references. `name` alone means `this.name`; `this` is a `Cat`, and access is allowed. In `k.name`, the type of `k` is `Cat`; that is allowed too. In `h.name`, the type of `h` is `Animal`, and compilation stops.

The reason for the rule lies in what `h` may refer to. `h` could be any `Animal`, for example a `Dog`. The permission `protected` gives is "a subclass may touch the part it inherited". A cat touching the internal data of a dog is outside this permission. That is why the compiler does not allow a reference of type `Animal`. For the same reason a parameter of type `Dog` does not work either: inside `Cat`, `k.name` with `k` a `Dog` again produces the message `name has protected access in Animal`.

Classes in the same package have no such restriction. There `protected` includes package access. A class `Vet` in package `animals` can read the `name` of any `Animal` and call `heal`, even though it is not a subclass.

4.3.5.4 Limiting inheritance with `final`

The `final` keyword is used to limit inheritance. If a method is declared `final`, no subclass can override it. If a class is declared `final`, no subclass can be derived from it.

```
class Animal {
    public final void breathe() { ... }
}
class Cat extends Animal {
    public void breathe() { ... }
}
```

```
error: breathe() in Cat cannot override
  breathe() in Animal
  overridden method is final
```

```
final class Fixed { ... }
class Test extends Fixed { }
```

```
error: cannot inherit from final Fixed
```



Figure 4.3.13: No subclass can be derived from a `final` class; a `final` method cannot be overridden in a subclass.

The Java library has many `final` classes: `String`, `Math` and `Integer` are among them. Writing `class MyString extends String { }` gives the error `cannot inherit from final String`. The reason is security and consistency. `String` objects must be immutable. When a method receives a `String`, it trusts that the text will not change later. File names, network addresses and `HashMap` keys, for example, rely on this trust. If `String` had a mutable subclass, this promise would be broken. `final` closes that door.

A `final` method is used for a similar reason. If the `breathe()` method of `Animal` is critical for the correctness of the health computation, no subclass should change it. Making the methods called from a constructor `final` also prevents an override that reads fields of the subclass that have not been initialized yet.

For variables, `final` carries another meaning: the variable is assigned once and becomes a constant. The same word carries three meanings in three places. What they share is “from now on, no change”: the value of the variable, the body of the method, the place of the class in the hierarchy.

4.3.5.5 Narrowing the view of an object

A subclass object can be assigned to a variable of the superclass type. This assignment is called **upcasting**. The assignment happens by itself, no cast operator is needed, because every `Tire` is a `Part`.

```
Part p = new Tire("Petlas", "XYZ-123", "225/45R17", true);
System.out.println(p.serialNumber());   // allowed → XYZ-123
// p.isWinter()                         // not allowed
```

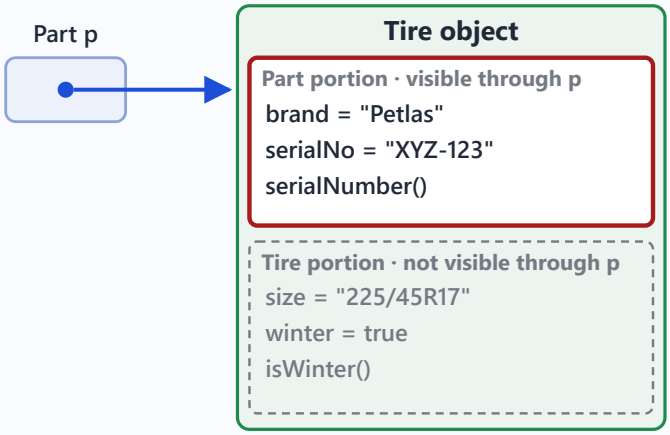


Figure 4.3.14: The reference `p` of type `Part` refers to a `Tire` object. Through `p` only the `Part` portion of the object is visible; `size`, `winter` and `isWinter()` stay hidden.

Figure 4.3.14 shows an important distinction. The object itself has not changed; it is still a `Tire` with all its fields. What has changed is only the width of the window through which it is seen. The compiler decides which methods can be called by looking at the **type of the variable**. The type of `p` is `Part`; `Part` has `serialNumber()` and does not have `isWinter()`:

```
error: cannot find symbol
  symbol:   method isWinter()
  location: variable p of type Part
```

This restriction is a design tool. If a method takes a `Part` as a parameter, it works with every kind of part and can use only operations specific to parts. That way only the needed face of the object is opened to the outside. A method that counts stock does not have to know whether a tire is a winter tire.

To reach a method specific to `Tire`, **downcasting** is used. A downcast is written explicitly and checked at run time. That is why the type is first checked with `instanceof`:

```
Part p = new Tire("Petlas", "XYZ-123", "225/45R17", true);
if (p instanceof Tire) {
    Tire l = (Tire) p;          // safe
    System.out.println(l.isWinter());
}
Part q = new Part("Bosch", "AB-9");
Tire l2 = (Tire) q;             // q is not a Tire
```

```
true
Exception in thread "main" java.lang.ClassCastException: class Part cannot be cast to
class Tire
```

Because `p instanceof Tire` is true, `(Tire) p` is safe and `true` is printed. `q` is a plain `Part`. The compiler accepts `(Tire) q`, because a `Part` variable may be referring to a `Tire`. At run time the JVM looks at the object, sees that it is not a `Tire`, and throws `ClassCastException`. A downcast does not produce a new object. It lets you look at the same object through a wider window; if the object does not fit that window, an exception occurs.

Common mistake

An unchecked downcast: writing `(Tire) p` causes `ClassCastException` if `p` is not a `Tire`. Before the cast, the check `if (p instanceof Tire)` is made. Since Java 16 the two can be written in one step: `if (p instanceof Tire l) { … l.isWinter() … }`. Here `l` holds the cast reference if the check is true.

4.3.6 Relationships between classes

Inheritance is not the only way to relate two classes. A class can also hold an object of another class as a field. This section compares the two relationships, explains why Java does not allow multiple inheritance, and shows with code traces two problems inheritance causes when it is misused.

4.3.6.1 The is-a and has-a relationships

Definition 4.3.7

The **is-a** (“is a kind of”) relationship is inheritance: a class is derived from another class and becomes a kind of it. A subclass object can also be used as an object of the superclass.

The **has-a** (“owns”) relationship is **composition**: a class contains a field of another class's type. The object holds references to other objects inside it.

The easiest way to tell the two relationships apart is to form a sentence. “A car is a vehicle” is a correct sentence; `Car extends Vehicle` is written. “A car is an engine” is wrong. The correct sentence is “a car has an engine”; the engine becomes a field of the car. Real classes often carry both relationships together:

```
// Is-a relationship (inheritance)
public class Car extends Vehicle {
    // Has-a relationship (composition)
    private Engine engine;

    public Car(int horsepower) {
        engine = new Engine(horsepower);
    }
    public void drive() {
        engine.start();   // delegate to Engine
        accelerate(50);         // inherited from Vehicle
        System.out.println("Speed: " + speed
                           + " km/h");
    }
}
// Engine: horsepower field and start():
//   "Engine started: " + horsepower + " HP"
```

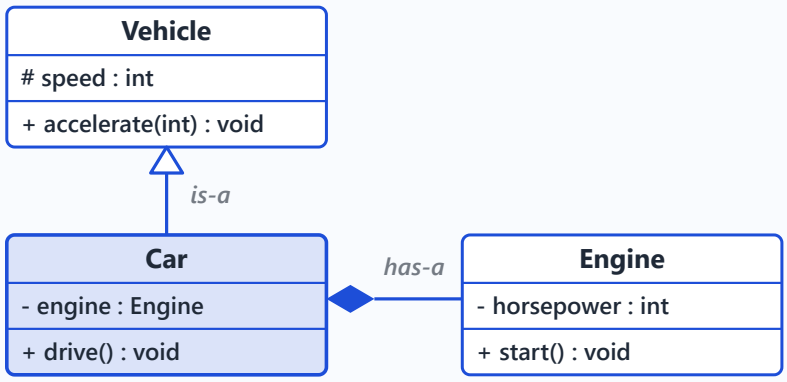


Figure 4.3.15: `Car` derives from `Vehicle` (is-a). `Car` contains an `Engine` object (has-a). In UML, composition is drawn with a filled diamond on the side of the containing class.

Class `Vehicle` has a `speed` field and an `accelerate(int)` method; `speed` is `protected` so that subclasses can reach it. Class `Engine` holds the horsepower, and its `start()` method prints “Engine started: … HP”. `Car` creates its own engine in its constructor. Its `drive()` method uses both relationships together: it delegates the job of starting the engine to the `Engine` object, and accelerates with the method it inherited from `Vehicle`. `new Car(110).drive();` prints:

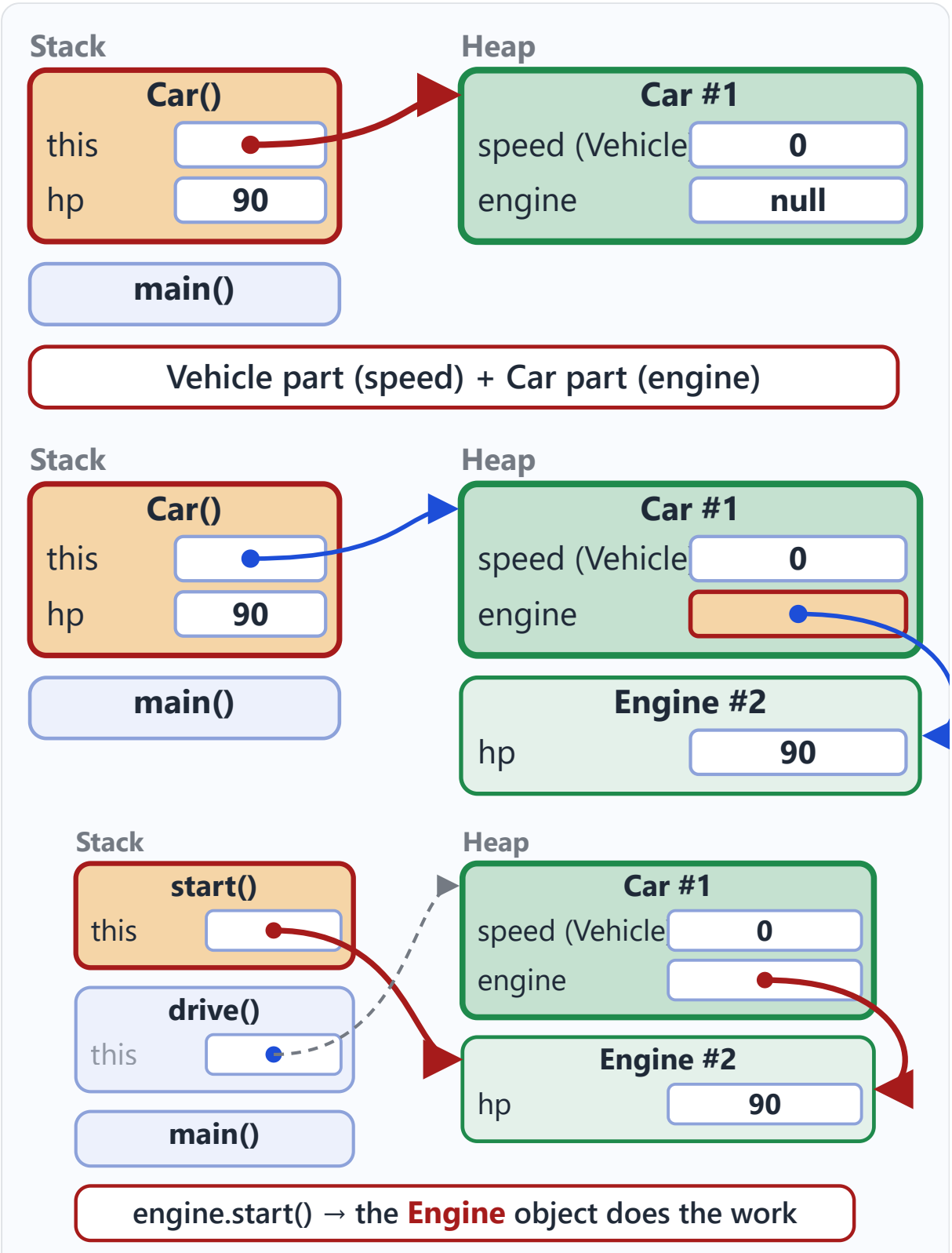
```
Engine started: 110 HP
Speed: 50 km/h
```

4.3.6.2 Code trace: car and engine

The difference between the two relationships in memory can be seen with a trace. The trace in the slides uses a shortened version of the same classes:

```
class Engine {
    private int hp;
    Engine(int hp) { this.hp = hp; }
    void start() {
        System.out.println("Engine: " + hp + " HP");
    }
}
class Car extends Vehicle {        // is-a
    private Engine engine;           // has-a
    Car(int hp) { engine = new Engine(hp); }
    void drive() {
        engine.start();  accelerate(50);
        System.out.println("Speed: " + speed);
    }
}
new Car(90).drive();   // main
```

```
Engine: 90 HP
Speed: 50
```


Stack
Heap
Car()
this
hp
90
Car #1
speed (Vehicle
0
engine
null
main()
Vehicle part (speed) + Car part (engine)
Stack
Heap
Car()
this
hp
90
Car #1
speed (Vehicle
0
engine
main()
Engine #2
hp
90
Stack
Heap
start()
this
drive()
this
main()
Car #1
speed (Vehicle
0
engine
Engine #2
hp
90
engine.start() → the Engine object does the work

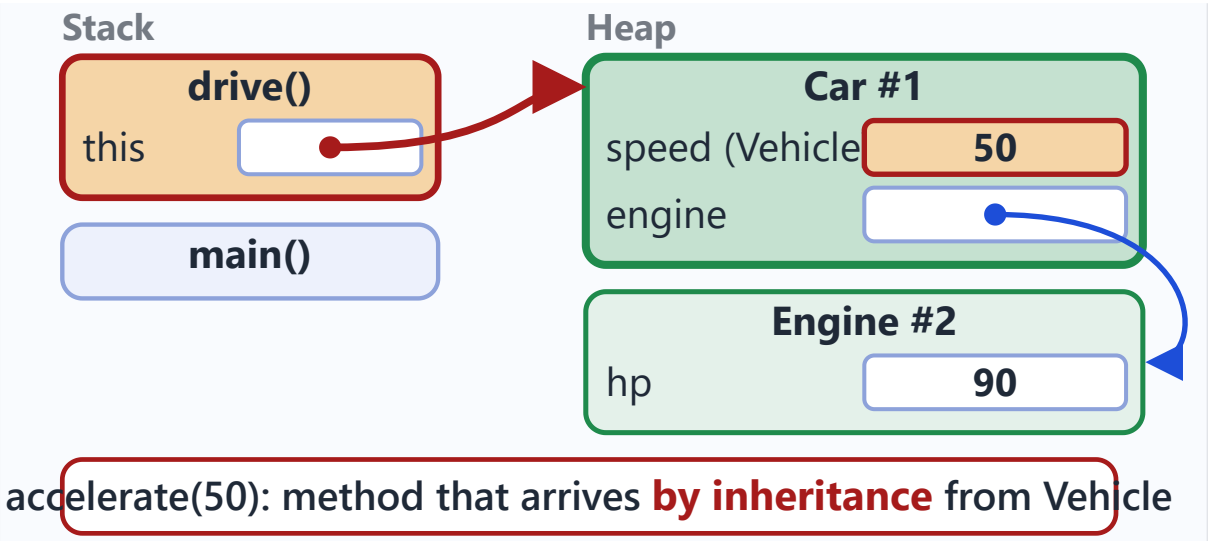


Figure 4.3.16: Four moments of the is-a and has-a trace. (a) `new Car(90)` creates a single object; the `speed` that comes from `Vehicle` is inside this object. (b) The constructor creates a second object, an `Engine`; the `engine` field refers to it. (c) `engine.start()`: the work is delegated to `Engine` by following the reference. (d) `accelerate(50)` sets the `speed` field of the same `Car` object to 50.

Figure 4.3.16 clearly separates what the two relationships look like in memory. Inheritance gives **a single object**. The `speed` field is not in a separate `Vehicle` object but in the `Car` object itself. Composition gives **two separate objects**. The `engine` field in the `Car` object is a reference to another object on the heap. A car is not an engine; it has one. The link is a reference.

The flow of the calls differs too. In the call `engine.start()`, `Car` does not do the work itself; it follows the reference and delegates to the `Engine` object. A `start()` frame is added to the stack, and the `this` in that frame refers to the `Engine` object. In the call `accelerate(50)`, `this` does not change; the method is written in `Vehicle` but works on the same `Car` object.

4.3.6.3 Multiple inheritance

In Java, a class can derive from only one class; multiple inheritance of classes is not directly supported:

```
class D extends B, C { }   // invalid
```

```
error: '{' expected
class D extends B, C { }
                 ^
```

The message looks a little odd. The compiler does not say “multiple inheritance is forbidden”; it reports a syntax error. After `extends` it expects a single class name, and right after the name it looks for the `{` of the class body. It stops when it sees the comma.

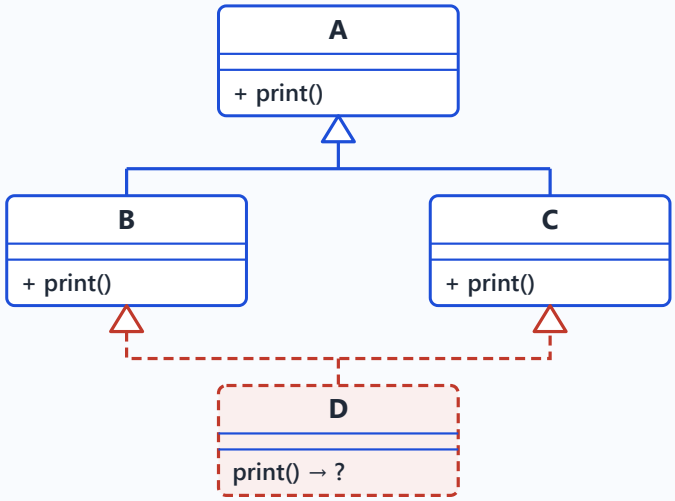


Figure 4.3.17: The diamond problem. If `D` derived from both `B` and `C`, and both overrode the `print()` method of `A` differently, it would be unclear which one `D` gets.

The reason for the ban is the **diamond problem**. Figure 4.3.17 shows the situation: `B` and `C` derive from `A`, and both override `print()` differently. If `D` could derive from both, which version would `d.print()` run? Because the diagram looks like a diamond, the problem has this name. The problem has a second face: should the fields of `A` be present once or twice in a `D` object? Languages that allow multiple inheritance, such as C++, answer these questions with complicated rules. The designers of Java chose not to take multiple inheritance of classes at all.

If the abilities of several classes are needed, there are two ways. The first is composition: the class holds as fields the objects whose abilities it wants to use. The second is **interfaces**. A class can extend a single class while implementing many interfaces:

```
interface Chargeable { void charge(); }
class ElectricCar extends Car
                    implements Chargeable {
    @Override
    public void charge() {
        System.out.println("Charging.");
    }
}
```

`ElectricCar` is both a `Car` and something that can be charged. The interface makes just one promise: there will be a method called `charge()`. The class writes the body. Because classic interfaces have no fields, the field face of the diamond problem does not arise.

4.3.6.4 Design principles: inheritance or composition?

Four principles guide design with inheritance.

- **The is-a test.** If the sentence "Is every X a Y?" is not true, inheritance is not used. A car *is not* an engine, it *has* one; the right relationship is composition.
- **Each class manages its own data.** The fields of the superclass are given values with `super(...)` and the methods of the superclass; the fields are kept `private`.
- **Override the general methods.** `toString` and `equals`, and with it `hashCode`, should be overridden in every meaningful class.
- **Shared concepts go to an abstract class.** If "Animal" alone should not be an object, that is, if nobody should create a vague "animal", the common concept of the subclasses is represented by an **abstract** class.

The choice between inheritance and composition is an application of the first principle. Table 4.3.9 compares the two ways on a few criteria.

	Inheritance (`extends`)	**Composition (object as a field)**
Relationship	is-a: "every X is a Y"	has-a: "X has a Y"
Coupling	Tight: the subclass depends on the internals of the superclass too	Loose: depends only on the `public` methods of the inner object
Reuse	Automatic: all `public` methods come along (even unwanted ones)	Selective: only the methods that should be forwarded are written
Change	The superclass is fixed at compile time	The field can be replaced with another object at run time
Use in place of the supertype	Direct: `Animal h = new Cat()`	If a common interface is implemented

Table 4.3.9: Comparison of inheritance and composition.

Inheritance is appropriate when the is-a relationship is really true and the superclass was designed and documented to be extended. `Cat` and `Animal` fit this case. Composition is appropriate when the goal is only to use the code of another class. A stack class is an example. `MyStack` can use an `ArrayList` to hold its elements. But a stack is *not* a list. If `MyStack extends ArrayList` is written, the stack also receives all the `public` methods of the list. Someone can insert an element into the middle of the stack with `add(0, x)`. This breaks the stack's rule "add and remove only at the top". The right way is to keep the list in a `private` field and expose only `push` and `pop`.

The Change row states an important difference too. Because the engine of a `Car` is a field, it can be replaced while the program runs. The gasoline engine is removed and an electric engine installed; if both implement a common `Engine` interface, the code of `Car` does not change. With inheritance, the superclass of `Car` is written in the source code and does not change once compiled.

4.3.6.5 Code trace: the fragile base class

The tight coupling of inheritance sometimes causes unexpected bugs. The example below has a list class. The goal is to write a subclass that counts the elements added to it.

```
class MyList {
    private String[] items = new String[10];
    private int size;
    void add(String s) { items[size++] = s; }
    void addAll(String[] arr) {
        for (String s : arr) add(s);
    }
    int size() { return size; }
}
class CountingList extends MyList {
    int addCount = 0;
    @Override void add(String s) {
        addCount++;
        super.add(s);
    }
    @Override void addAll(String[] arr) {
        addCount += arr.length;
        super.addAll(arr);
    }
}
CountingList l = new CountingList();   // main
l.addAll(new String[] {"a", "b", "c"});
System.out.println(l.addCount + " " + l.size());
```

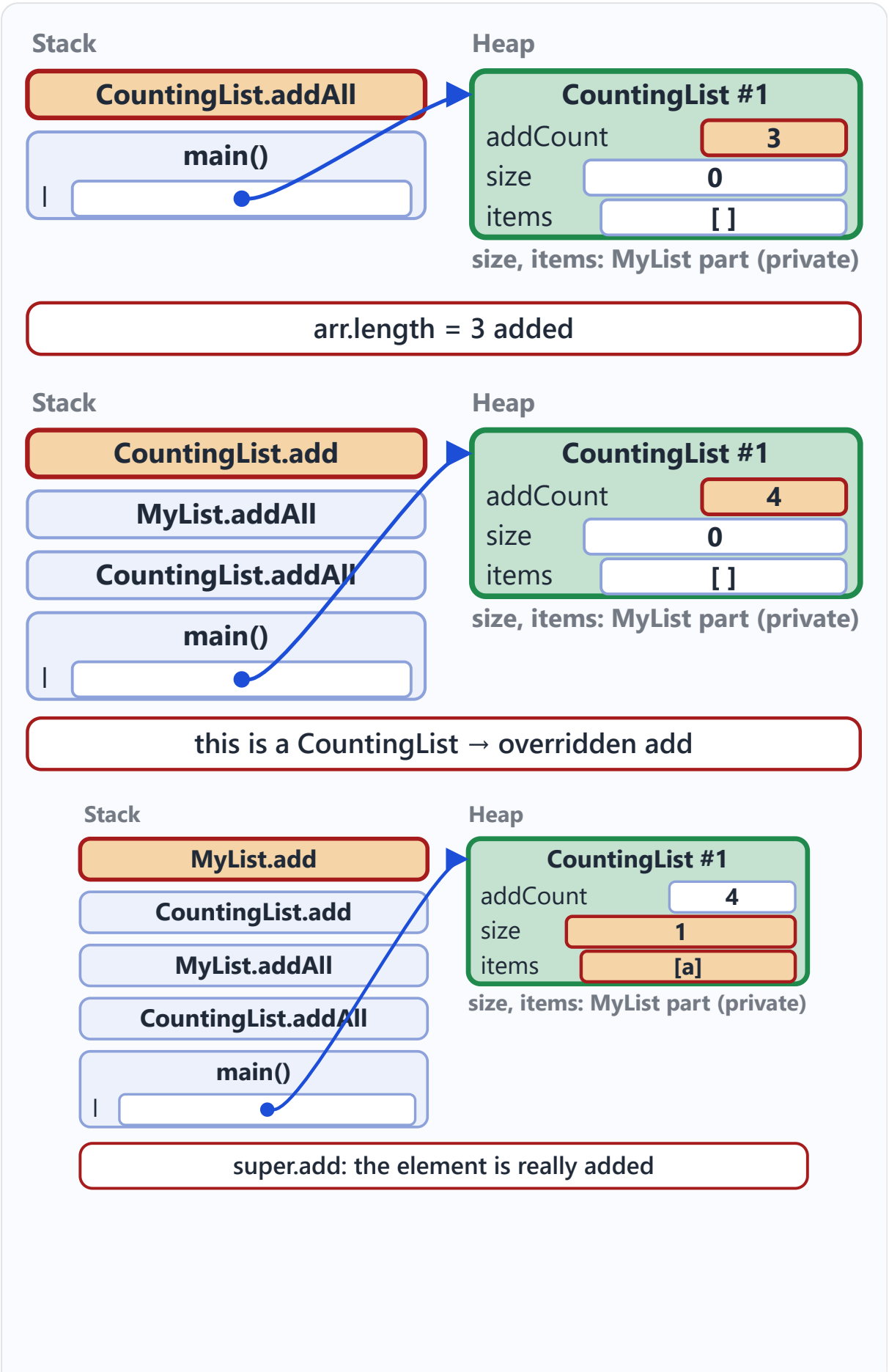
```
6 3
```

Three elements were added and the list really holds three, but the counter says 6. Both methods of `CountingList` look right on their own. `add` increases the counter by one and leaves the work to the superclass. `addAll` adds the length of the array to the counter and leaves the work to the superclass. Table 4.3.10 shows the source of the bug step by step.

Step	Method running	Event	addCount	size
1	`CountingList.addAll`	The object is a `CountingList`; the version in the subclass is called.	0	0
2	`CountingList.addAll`	`addCount += arr.length`	3	0
3	`MyList.addAll`	`super.addAll(arr)`; the loop calls `add("a")`, that is `this.add("a")`.	3	0
4	`CountingList.add`	`this` is a `CountingList`: the overridden `add` runs and the counter goes up.	4	0

Step	Method running	Event	addCount	size
5	`MyList.add`	`super.add(s)` writes the element into the array.	4	1
6	(loop)	Steps 3–5 repeat for "b" and "c".	6	3
7	`main`	`println`: wrong result	6	3

Table 4.3.10: Trace of `l.addAll({"a", "b", "c"})`.


Stack
Heap
CountingList.addAll
main()
CountingList #1
addCount
3
size
0
items
[ ]
size, items: MyList part (private)
arr.length = 3 added
Stack
Heap
CountingList.add
MyList.addAll
CountingList.addAll
main()
CountingList #1
addCount
4
size
0
items
[ ]
size, items: MyList part (private)
this is a CountingList → overridden add
Stack
Heap
MyList.add
CountingList.add
MyList.addAll
CountingList.addAll
main()
CountingList #1
addCount
4
size
1
items
[a]
size, items: MyList part (private)
super.add: the element is really added

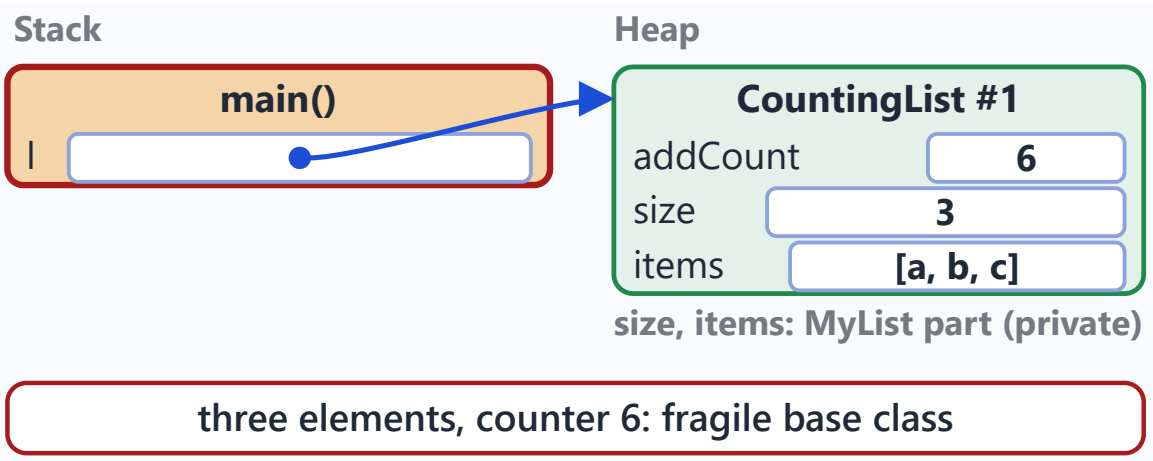


Figure 4.3.18: Four moments of the fragile base class trace. (a) `addAll` of the subclass adds 3 to the counter. (b) The `add(s)` in the superclass's loop returns to the subclass because the object is a `CountingList`; the same element is counted a second time. (c) `super.add` writes the element: `size` = 1. (d) The result is 6 3.

The bug is born in the third step. `MyList.addAll` does its work by calling `add(s)` for each element. This is an **implementation detail** of `MyList`; it may not be written in the class's documentation. The `add(s)` in the code of `MyList` is really `this.add(s)`, and `this` is a `CountingList`. The call goes to the overridden version. Each element is counted once in `addAll` and once again in `add`.

Definition 4.3.8

The **fragile base class** problem is that the correctness of a subclass depends on the internals of its superclass. Even a change in the superclass that does not alter its externally visible behavior can break the subclass.

Here is the fragility side of the problem. The programmer of `CountingList` may notice the bug and delete the override of `addAll`. The counter is then correct, because each element is counted only in `add`. But this fix depends on the code `MyList` has today. If a new version of `MyList` writes `addAll` to copy the array directly for speed, `add` is never called and the counter stays at 0. In both cases the behavior `MyList` offers its users is the same, but `CountingList` breaks.

Example 4.3.6

On a new `CountingList`, first `add("x")` and then `addAll(new String[] {"y", "z"})` are called. What are `addCount` and `size()`?

`add("x")` runs the subclass's `add` directly: counter 1, size 1. `addAll` first adds 2 to the counter: counter 3. Then the loop of the superclass calls `add` twice and each goes back to the subclass: counter 5, size 3. The result is **5 3**. Had only `add` been used, the result would have been right; the bug comes from overriding `addAll` and `add` together.

4.3.6.6 The composition fix: forwarding

The clean fix for the fragile base class problem is to give up inheritance and move to composition:

```
class CountingList2 {
    private final MyList list = new MyList(); // has-a
    private int addCount = 0;
    void add(String s) {
        addCount++;
        list.add(s);            // forwards
    }
    void addAll(String[] arr) {
        addCount += arr.length;
        list.addAll(arr);  // forwards
    }
    int getAddCount() { return addCount; }
    int size() { return list.size(); }
}
```

```
CountingList2 l2 = new CountingList2();
l2.addAll(new String[] {"a", "b", "c"});
System.out.println(l2.getAddCount()
    + " " + l2.size());
```

```
3 3
```

Definition 4.3.9

Forwarding is passing a call that arrives at the composing class on to a method of the inner object. Each method of the outer class first does its own work, then forwards the call to the inner object.

The new class does not derive from `MyList`; it holds a `MyList` inside. The `add` calls inside `list.addAll(...)` run on a plain `MyList` object. The `this` of that object is a `MyList`; the call does not come back to `CountingList2`. The counter is right: 3. Even if the internal detail of `MyList` changes, the result does not. `CountingList2` depends only on the `public` methods of `MyList`.

This solution has a price. `CountingList2` is not a `MyList`; it cannot be given where a `MyList` is expected. If needed, `MyList` and `CountingList2` implement a common interface and the code is written against that interface type. The forwarding approach is common in the Java library too. For example, `Collections.unmodifiableList(...)` returns an object that wraps a list and forwards read calls to it.

4.3.6.7 The substitution principle

The is-a test can be misleading when it is applied in everyday language. In mathematics every square is a rectangle. Based on this sentence, writing `Square extends Rectangle` seems natural:

```
class Rectangle {
    protected int width, height;
    void setWidth(int width) { this.width = width; }
    void setHeight(int height) { this.height = height; }
    int area() { return width * height; }
}
class Square extends Rectangle {
    @Override void setWidth(int k) { width = k; height = k; }
    @Override void setHeight(int k) { width = k; height = k; }
}
static void widen(Rectangle d) {
    d.setWidth(5);
    d.setHeight(4);
    System.out.println("Expected 20, got "
        + d.area());
}
// in main:
widen(new Rectangle());
widen(new Square());
```

```
Expected 20, got 20
Expected 20, got 16
```

Definition 4.3.10

The **Liskov substitution principle** (LSP): code written against a superclass type must also work correctly with an object of a subclass. It is not enough for the code to compile; the behavior must stay correct as well.

The method `widen` sets the width of a rectangle to 5 and its height to 4 and expects the area to be 20. With a plain `Rectangle` the expectation holds. With a `Square`, `setWidth(5)` sets both sides to 5, and then `setHeight(4)` sets both sides to 4. The area comes out as 16. The code compiled, no exception occurred, but the result is wrong.

Where is the bug? `widen` is not faulty. Class `Rectangle` makes the promise “width and height change independently of each other”: `setWidth` does not touch the height. `widen` relied on this promise. `Square` cannot keep it, because the two sides of a square cannot be independent. The fault lies with the subclass that broke the promise. The mathematical relationship is true, but the relationship between the classes is wrong.

There are two common solutions to this problem. The first is not to derive `Square` from `Rectangle`; both derive from a common `Shape` and each writes its own `area()`. The second is to make the classes immutable. Width and height are given only in the constructor, and there are no `setWidth` and `setHeight` at all. Then there is no promise like "changing the width leaves the height alone", and a square can really behave like a rectangle.

The lesson is this: the is-a test is applied not to the real world but to the **behavior** of the class. The right question is not "is a square a rectangle?" but "can `Square` answer every method of `Rectangle` the way a `Rectangle` does?"

Example 4.3.7

After the lines `Rectangle d = new Square(); d.setWidth(3); d.setHeight(7);`, what is `d.area()`?

The type of the variable is `Rectangle`, and the object is a `Square`. The calls go to the overridden versions. `setWidth(3)`: width = height = 3. `setHeight(7)`: width = height = 7. The area is 7 × 7 = **49**. The last call determines both sides. With a plain rectangle the result would be 3 × 7 = 21.

4.3.7 The `Object` class

Every class in Java derives from `Object`, whether or not it is written. This section describes the methods `Object` gives every object and how to override the three most important of them, `toString`, `equals` and `hashCode`, correctly.

4.3.7.1 The ancestor of every class

Every class derives from `java.lang.Object`. If `extends` is not written, this happens implicitly:

```
class Point { }
// what the compiler sees:
class Point extends Object { }
```

That is why some methods are ready on every object. Table 4.3.11 gives the four most used. The default versions of these methods are general-purpose and often useless. In meaningful classes the first three are overridden.

Method	Default behavior
`String toString()`	The class name + `@` + the hexadecimal form of the `hashCode` value
`boolean equals(Object o)`	Compares references: `this == o`

Method	Default behavior
`int hashCode()`	An integer specific to the object (usually different for each object)
`Class getClass()`	The runtime class of the object (`final` ; cannot be overridden)

Table 4.3.11: The most commonly used methods of the `Object` class.

`Object` has other methods too: `wait` , `notify` and `notifyAll` , used with threads, and `clone` , used to copy objects. They are not the subject of these notes. A variable of type `Object` can refer to any object. That is why the parameter of `equals` is `Object` : an object must be comparable with any other object.

4.3.7.2 The `toString` method

`public String toString()` shows an object as text. `System.out.println(obj)` and concatenation of the form `"..." + obj` call this method automatically. If it is not overridden, a text containing the class name and the `hashCode` value of the object is returned:

```
class Point {
    private int x, y;
    Point(int x, int y) {
        this.x = x; this.y = y;
    }
}
// in main
System.out.println(new Point(3, 4));
```

```
Point@28a418fc
```

The `28a418fc` after the `@` sign is the hexadecimal form of the `hashCode()` value. The value may differ on every run or on every JVM. It says nothing about the contents of the object; it is useless even when debugging. Once overridden, the object is printed readably:

```
class Point {
    private int x, y;
    Point(int x, int y) {
        this.x = x; this.y = y;
    }
    @Override
    public String toString() {
        return "Point[x=" + x + ", y=" + y + "]";
    }
}
// in main
Point n = new Point(3, 4);
System.out.println(n);
System.out.println("Position: " + n);
```

```
Point[x=3, y=4]
Position: Point[x=3, y=4]
```

On the second line, because one side of the `+` operator is a `String`, Java converts the other side to text too, and calls `toString()` for it. The return type of `toString()` is `String` and its access is `public`. Forgetting to write `public` when overriding gives the access-narrowing error seen in Table 4.3.6.

There is one more detail. If a `null` reference is printed, `toString` is not called; "null" is printed:

```
Point n = null;
System.out.println(n);
System.out.println("Position: " + n);
```

```
null
Position: null
```

`println` and string concatenation treat `null` specially. But had `n.toString()` been called explicitly, a `NullPointerException` would occur; a method cannot be called on a `null` reference.

4.3.7.3 `==` or `equals`?

There are two ways to compare two objects, and they ask different questions. `==` asks, "do these two references point to the same object?" `equals` asks, "are these two objects equivalent?" In the `Person` class below, let `equals` be overridden to compare name and age:

```
Person k1 = new Person("Ali", 25);
Person k2 = new Person("Ali", 25);
Person k3 = k1;
System.out.println(k1 == k2);
System.out.println(k1 == k3);
System.out.println(k1.equals(k2));
```

```
false
true
true
```

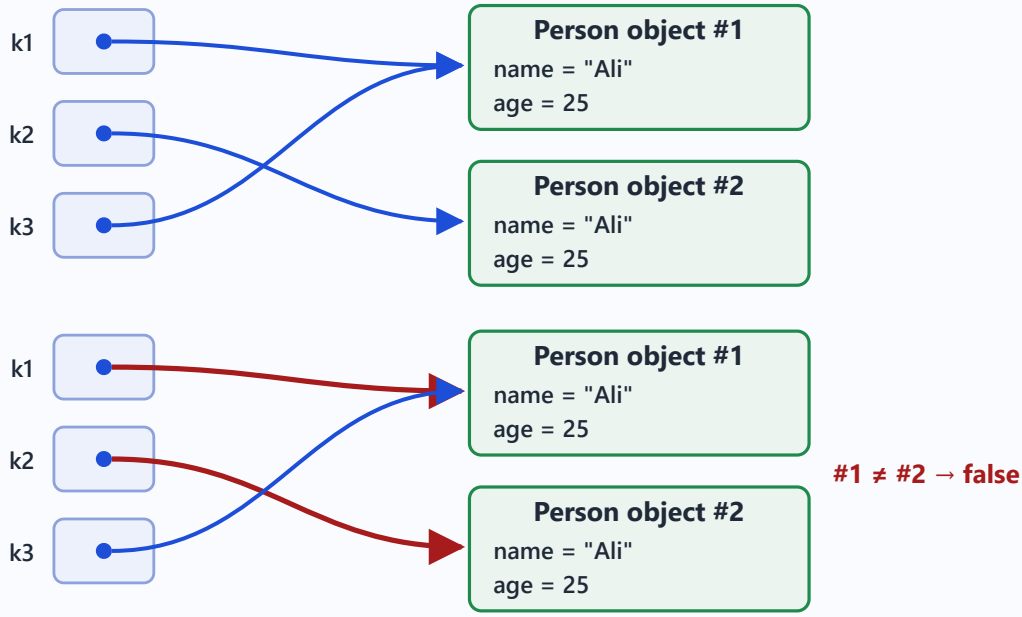


Figure 4.3.19: Two moments of the `==` and `equals` trace. (a) `k3 = k1` only copies the address; `k1` and `k3` point to the same object. (b) `k1 == k2` : two separate objects, the result is `false` .

Let us follow the lines in order. The first two `new` expressions create two separate objects with the same content on the heap; `k1` and `k2` hold their addresses. `k3 = k1` does not create a new object; it only copies the address. Now `k1` and `k3` point to the same object (Figure 4.3.19). `k1 == k2` compares two different addresses and gives `false` . `k1 == k3` compares the same address and gives `true` . `k1.equals(k2)` compares the contents: both objects have the name "Ali" and the age 25, so the result is `true` .

Had `equals` not been overridden, the last line would give `false` . The `equals` of `Object` is just `this == o` . For `String` comparison the rule is the same: write `s1.equals(s2)` , not `s1 == s2` . `new String("Java") == new String("Java")` gives `false` , and `equals` gives `true` .

4.3.7.4 The `equals` contract

`public boolean equals(Object other)` says whether two objects are equivalent. By default it only compares references; if it is not overridden, it behaves like `==`. Once overridden, the method compares the state of the two objects, that is, their fields, and returns `true` if they are equal.

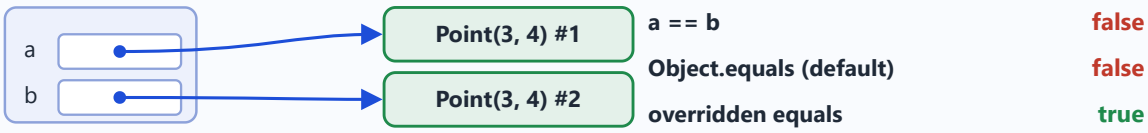


Figure 4.3.20: Two separate objects with the same content. `==` and the default `equals` compare addresses and give `false`. An overridden `equals` compares the fields and gives `true`.

`equals` cannot be written arbitrarily. The Java library, collections in particular, assumes that `equals` obeys certain rules. These rules are called the `equals` contract (Table 4.3.12).

Property	Rule
Reflexivity	`x.equals(x)` → `true`
Symmetry	`x.equals(y)` ⇔ `y.equals(x)`
Transitivity	if `x` = `y` and `y` = `z`, then `x` = `z`
Consistency	The result does not change as long as the fields do not change
null	`x.equals(null)` → `false`

Table 4.3.12: The `equals` contract (`x`, `y`, `z` are non-`null` references).

These rules are the properties of equality in mathematics. When they are broken, the bugs appear indirectly. If symmetry is broken, `list.contains(x)` may give `true` in one order and `false` in the other. If consistency is broken, an object added to a `HashSet` cannot be found later.

4.3.7.5 `equals` with `instanceof`

The parameter of `equals` has type `Object`. The object that arrives may not be a `Point`; it may be a string, a person or `null`. That is why a type check is made before moving on to the fields:

```
public boolean equals(Object obj) {
    if (obj instanceof Point) {
        // the object is a Point;
        // cast and compare
        Point other = (Point) obj;
        return x == other.x && y == other.y;
    } else {
        // the object is not a Point;
        // it cannot be equal
        return false;
    }
}
```

The method first asks with `obj instanceof Point` whether the incoming object is a `Point`. If so, it downcasts with `(Point) obj`. The cast is safe only after the type check. Then it compares the two coordinates. If the object is not a `Point`, it cannot be equal and `false` is returned.

```
Point a = new Point(3, 4);
Point b = new Point(3, 4);
System.out.println(a == b);
System.out.println(a.equals(b));
System.out.println(a.equals("3,4"));
System.out.println(a.equals(null));
```

```
false
true
false
false
```

The first line compares two separate objects with `==`: `false`. The second compares the contents: `true`. In the third, `"3,4"` is not a `Point` but a `String`; `instanceof` gives false and `false` is returned. Since the cast is never reached, no `ClassCastException` occurs either. In the fourth, `null instanceof Point` is false. `instanceof` always gives `false` for `null`. That is why an `equals` that uses `instanceof` needs no separate `null` check.

The parameter type must be `Object`. Writing `equals(Point n)` is overloading, not overriding. This mistake escapes the compiler and has insidious results:

```
class PointA {
    int x, y;
    PointA(int x, int y) { this.x = x; this.y = y; }
    public boolean equals(PointA n) {   // overloading!
        return x == n.x && y == n.y;
    }
}
// in main:
PointA p = new PointA(1, 2), q = new PointA(1, 2);
Object qo = q;
System.out.println(p.equals(q) + " " + p.equals(qo));
List<PointA> list = new ArrayList<>();
list.add(p);
System.out.println(list.contains(q));
```

```
true false
false
```

`p.equals(q)` gives the right result, because the compile-time type of `q` is `PointA` and the compiler chooses `equals(PointA)`. In `p.equals(qo)`, however, the type of the argument is `Object`. The compiler chooses `equals(Object)` from `Object`, which compares addresses: `false`. Library classes always call `equals(Object)`. That is why `ArrayList.contains` cannot find a point with the same content. With `@Override` written, the error would have been caught at compile time: `method does not override or implement a method from a supertype`.

4.3.7.6 `equals` with `getClass()`, and a trace

The second way to check the type is `getClass()`:

```
public class Person {
    private String name;
    private int age;
    // ... constructor ...

    @Override
    public boolean equals(Object obj) {
        // equal if the same reference
        if (this == obj) return true;
        // null or a different class: not equal
        if (obj == null
                || getClass() != obj.getClass())
            return false;
        // compare the fields
        Person person = (Person) obj;
        return age == person.age
               && name.equals(person.name);
    }
}
```

This pattern has three steps. First `this == obj` is checked. If it is the same object, there is no need to look at the fields and `true` is returned at once; this is a quick shortcut. Then `obj == null` and the class check are made. `getClass()` gives the actual class of the object; the classes of both sides must be exactly the same. Finally the fields are compared. Primitive types (`int age`) are compared with `==`, fields of object type (`String name`) with `equals`.

The two patterns behave differently for subclass objects (Table 4.3.13).

Type check	Can a subclass object count as equal?
`instanceof`	Yes (a `Cat` object is an `Animal`)
`getClass()`	No (the classes must be exactly the same)

Table 4.3.13: Type checking with `instanceof` and with `getClass()`.

```
class Student extends Person {         // Person's equals uses getClass()
    Student(String name, int age) { super(name, age); }
}
// in main:
System.out.println(new Person("Ali", 20).equals(new Student("Ali", 20)));
```

```
false
```

Because the `equals` of `Person` uses `getClass()`, it does not count a `Student` with the same name and age as equal. With `instanceof` it would give `true`. Which one is right depends on the design. `getClass()` preserves symmetry by itself. `instanceof` is safe as long as subclasses do not rewrite `equals`; if a subclass overrides `equals` with new fields, symmetry can break.

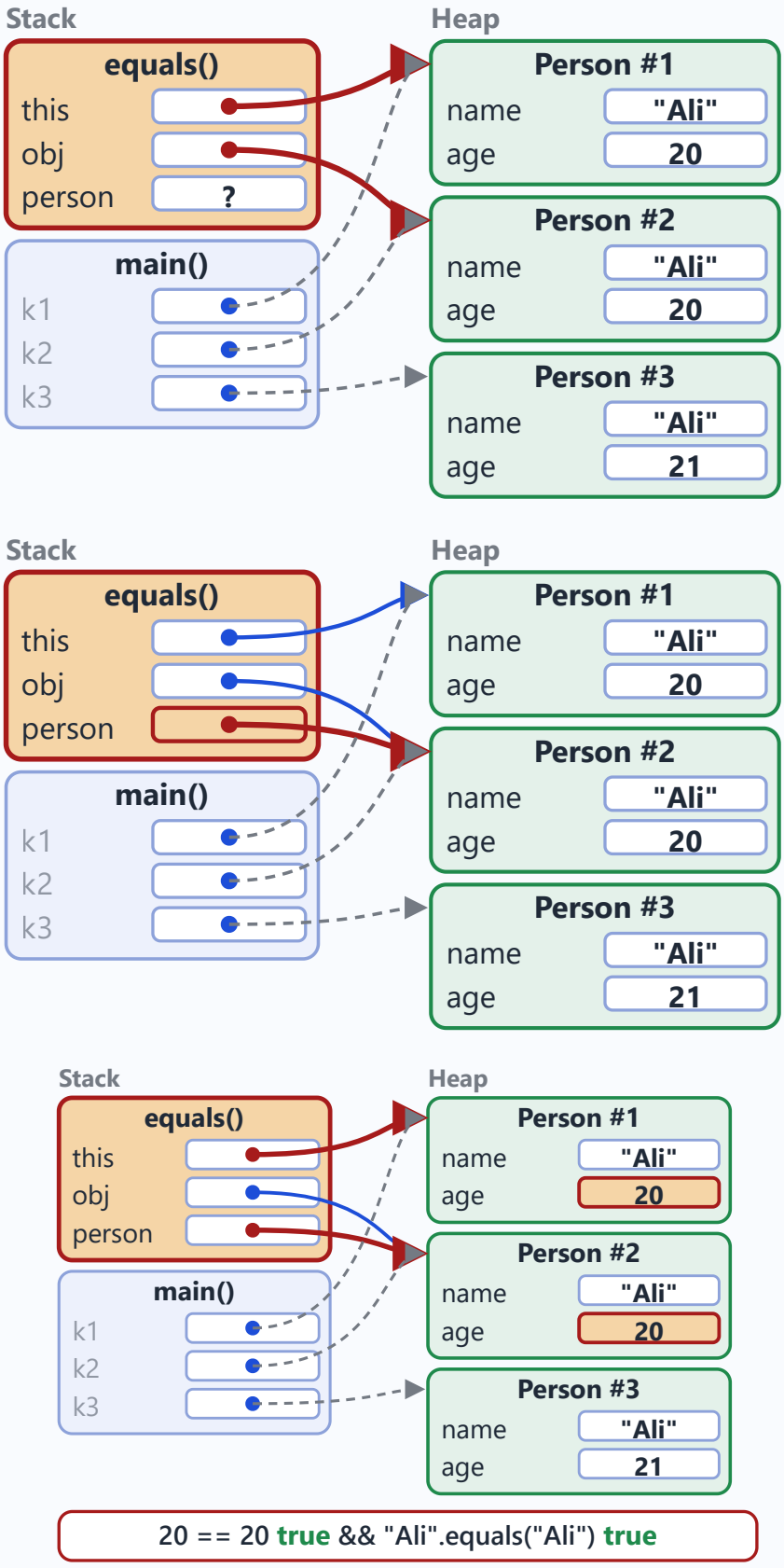
Now let us run the `equals` with `getClass()` through a trace:

```
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null || getClass() != obj.getClass())
        return false;
    Person person = (Person) obj;
    return age == person.age && name.equals(person.name);
}
// in main:
Person k1 = new Person("Ali", 20);
Person k2 = new Person("Ali", 20);
Person k3 = new Person("Ali", 21);
System.out.println(k1.equals(k2));
System.out.println(k1.equals(k3));
System.out.println(k1.equals(null));
```

```
true
false
false
```

Call	`this == obj`	`null` / class check	`age == person.age`	`name.equals(...)`	Result
`k1.equals(k2)`	#1 ≠ #2: false	passed (`Person` = `Person`)	20 == 20: true	"Ali" = "Ali": true	`true`
`k1.equals(k3)`	#1 ≠ #3: false	passed	20 == 21: false	not called (short circuit)	`false`
`k1.equals(null)`	false	`obj == null` is true; `getClass` is not called	—	—	`false`

Table 4.3.14: Trace of three `equals` calls. #1, #2 and #3 are the addresses of the objects.


Stack
equals()
this
obj
person
?
main()
k1
k2
k3
Heap
Person #1
name
"Ali"
age
20
Person #2
name
"Ali"
age
20
Person #3
name
"Ali"
age
21
Stack
equals()
this
obj
person
main()
k1
k2
k3
Heap
Person #1
name
"Ali"
age
20
Person #2
name
"Ali"
age
20
Person #3
name
"Ali"
age
21
Stack
equals()
this
obj
person
main()
k1
k2
k3
Heap
Person #1
name
"Ali"
age
20
Person #2
name
"Ali"
age
20
Person #3
name
"Ali"
age
21
20 == 20 true && "Ali".equals("Ali") true

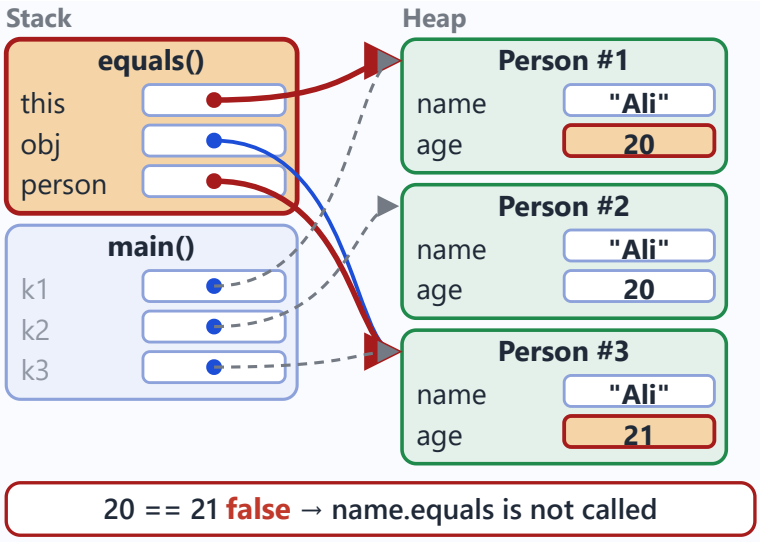


Figure 4.3.21: Four moments of the `equals` trace. (a) `k1.equals(k2)` : in the new frame `this` = #1, `obj` = #2. (b) The downcast does not produce a new object; `person` refers to the same #2 with type `Person` . (c) Age and name are equal: `true` . (d) `k1.equals(k3)` : the ages differ and `&&` short-circuits.

Three details of the trace matter. First, the type of `obj` is `Object` . `obj.age` cannot be written, because `Object` has no field called `age` . The downcast gives the reference `person` , which looks at the same object through a `Person` window (Figure 4.3.21, b). Second, `&&` short-circuits. If the ages differ, `name.equals` is never called. That is why writing the cheap comparison first is a good habit. Third, `||` short-circuits too. If `obj == null` is true, the right side is not evaluated. Had the order been reversed (`getClass() != obj.getClass() || obj == null`), `obj.getClass()` would run with a `null` argument and a `NullPointerException` would occur.

In terms of cost, `equals` compares a fixed number of fields. If one of the fields is a `String` , its `equals` does work proportional to the length of the string. The `this == obj` shortcut skips this work completely in the common case of an object being compared with itself.

4.3.7.7 `equals` and `hashCode`

Overriding `equals` carries an obligation. The rule is this: if `equals` is overridden, `hashCode` must be overridden too. Equal objects must have the same `hashCode` values.

Definition 4.3.11

A **hash code** is the integer that the `hashCode()` method returns for an object. Its contract: if `a.equals(b)` is true, then `a.hashCode() == b.hashCode()` must hold. The converse is not required; two objects that are not equal may have the same hash code.

```
import java.util.Objects;
public class Point {
    private int x, y;
    public Point(int x, int y) {
        this.x = x; this.y = y;
    }
    @Override public boolean equals(Object o) {
        return (o instanceof Point n)
               && x == n.x && y == n.y;
    }
    @Override public int hashCode() {
        return Objects.hash(x, y);
    }
    @Override public String toString() {
        return "Point[x=" + x + ", y=" + y + "]";
    }
}
```

```
HashSet<Point> set = new HashSet<>();
set.add(new Point(3, 4));
set.add(new Point(3, 4));
System.out.println(set.size() + " "
    + set.contains(new Point(3, 4)));
```

```
1 true
```

This `Point` class overrides all three methods. In `equals`, the form `o instanceof Point n` is used. This is **pattern matching**, available since Java 16: if the check is true, `n` holds the cast reference. The classic form is `if (o instanceof Point) { Point n = (Point) o; … }`. `hashCode` produces a value from the fields with `java.util.Objects.hash(x, y)`.

Why are the two methods needed together? When `HashSet` and `HashMap` look for an element, they first choose a "bucket" with `hashCode`, then call `equals` only against the elements in that bucket. If `hashCode` differs, equal objects fall into different buckets and are never compared. Table 4.3.15 compares the two cases.

In class `Point`	Output	
only `equals`	`2 false`	✗
`equals` + `hashCode`	`1 true`	✓

Table 4.3.15: Output of `set.size() + " " + set.contains(new Point(3, 4))`.

When only `equals` is written, the `hashCode` of `Object` is used, and in practice it gives two separate objects different values. The second `new Point(3, 4)` falls into another bucket. The set does not compare it with the first and adds it a second time: size 2. `contains` looks into a third bucket and does not find it: `false`. There is no compile error; the compiler does not check that the two methods agree. The bug shows only at run time, with a wrong result.

> **Common mistake**
>
> Overriding `equals` and forgetting `hashCode`. The class works without problems in an `ArrayList`, because `ArrayList.contains` uses only `equals`. The bugs start when the same objects are put in a `HashSet` or used as the key of a `HashMap`: duplicate elements, keys that cannot be found.

4.3.7.8 Code trace: inside a `HashSet`

How buckets are chosen can be followed with concrete numbers. The code below uses the classic `equals` form:

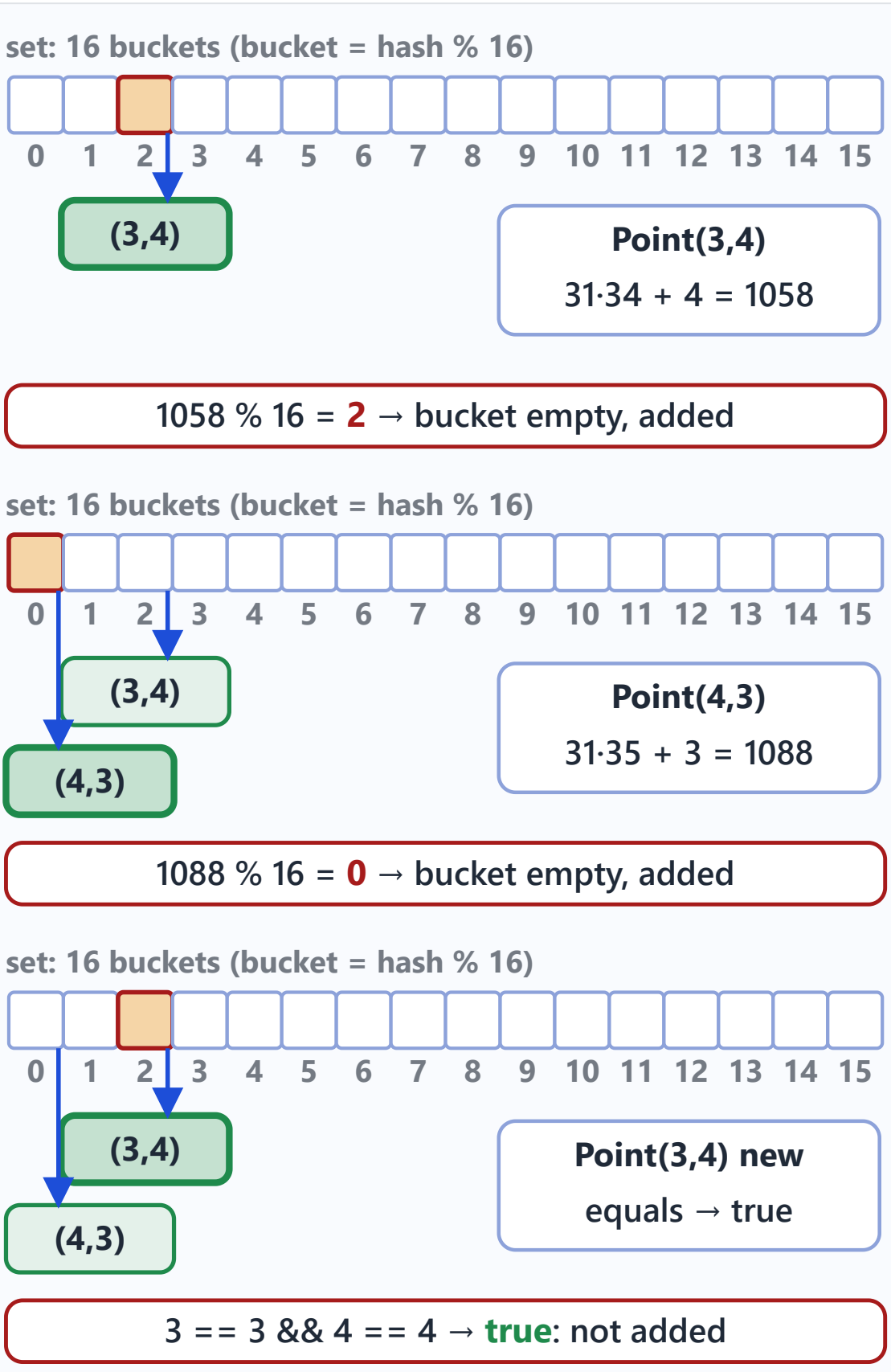
```
public boolean equals(Object o) {
    if (!(o instanceof Point)) return false;
    Point n = (Point) o;
    return x == n.x && y == n.y;
}
public int hashCode() {
    return Objects.hash(x, y);  // 31·(31 + x) + y
}
// in main:
HashSet<Point> set = new HashSet<>();
set.add(new Point(3, 4));
set.add(new Point(4, 3));
set.add(new Point(3, 4));
System.out.println(set.size());
System.out.println(set.contains(new Point(4, 3)));
```

```
2
true
```

`Objects.hash(x, y)` computes *31 · (31 + x) + y* for two fields. An empty `HashSet` starts with 16 buckets. An element's bucket is *hash % 16* for small hash values. (The library actually mixes the upper bits of the hash value into the lower bits and takes *hash & 15*. For positive values below 65,536 the result is the same as *hash % 16*.)

Operation	hash	bucket	What is in the bucket?	Result	size
`add(Point(3, 4))`	31 · 34 + 4 = 1058	2	empty	added	1
`add(Point(4, 3))`	31 · 35 + 3 = 1088	0	empty	added	2
`add(Point(3, 4))` (new object)	1058	2	(3, 4); `equals` is true	not added, `add` returns `false`	2
`contains(Point(4, 3))`	1088	0	(4, 3); `equals` is true	`true`	2

Table 4.3.16: Trace of the `HashSet`. Hash = 31 · (31 + x) + y, bucket = hash % 16.


set: 16 buckets (bucket = hash % 16)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(3,4)
Point(3,4)
31·34 + 4 = 1058
1058 % 16 = 2 → bucket empty, added
set: 16 buckets (bucket = hash % 16)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(3,4)
(4,3)
Point(4,3)
31·35 + 3 = 1088
1088 % 16 = 0 → bucket empty, added
set: 16 buckets (bucket = hash % 16)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(3,4)
(4,3)
Point(3,4) new
equals → true
3 == 3 && 4 == 4 → true: not added

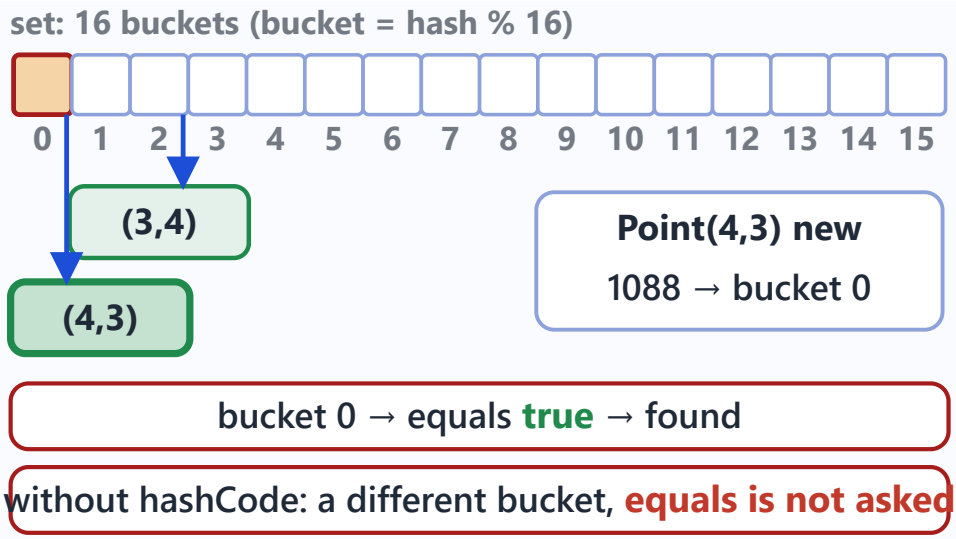


Figure 4.3.22: Four moments of the `HashSet` trace. (a) (3, 4): hash 1058, bucket 2. (b) (4, 3): hash 1088, bucket 0. (c) A new (3, 4) again falls into bucket 2; since `equals` is true, it is not added. (d) `contains` finds the bucket first, then asks `equals` : `true` .

Table 4.3.16 shows two important points. First, the same numbers in a different order give a different hash: 1058 for (3, 4), 1088 for (4, 3). That is why the formula has the factor 31; plain addition (x + y) would send the two points to the same value. Second, the set looks into only one bucket per operation. Searching among elements spread over 16 buckets is much faster than walking through all the elements one by one. With a well-spread `hashCode` , `add` and `contains` take O(1) on average.

Different objects can fall into the same bucket; this is called a **collision**. The hash of the point (1, 2) is $31 \cdot 32 + 2 = 994$, and $994 \% 16 = 2$. This point falls into the same bucket as (3, 4). In this case the set asks `equals` , gets the answer false, and adds the new point as the second element of the same bucket. Even the hash values can be equal: the hash of the point (2, 65) is 1088, like that of (4, 3). The contract allows this; equal hashes do not mean equal objects. `equals` has the last word.

Example 4.3.8

Let the `hashCode` of a class return 42 for every object, with `equals` written correctly. After adding (1, 2), (1, 2) and (2, 1) to a `HashSet` , what is the size? Does the set work correctly?

The contract is not broken: equal objects have the same hash, because all objects have the same hash. All elements fall into the same bucket. The second (1, 2) is caught by `equals` and not added; (2, 1) is added because it is not equal. The size is **2**, and `contains` answers correctly. But every operation has to walk through all the elements of that single bucket. The set works correctly, but takes O(n) instead of O(1) on average. A bad `hashCode` breaks not correctness but speed.

4.3.8 Common mistakes

Mistakes related to inheritance fall into two groups. Compile errors show up at once with javac's messages. Learning to read the messages lets you fix most of these errors in minutes. Logic errors pass the compiler and are noticed only when the program gives a wrong result. The two tables below gather the mistakes treated separately in the body of these notes. The messages were obtained with the javac and java tools of JDK 21.

Mistake	javac message	Fix
Not calling `super(...)` when the superclass has no no-argument constructor	`constructor Animal in class Animal cannot be applied to given types`	`super(name);` on the first line
Not writing `super(...)` on the first line; writing it together with `this(...)`	`call to super must be first statement in constructor`	A single `this` / `super` call, on the first line
Accessing a `private` field from a subclass	`health has private access in Animal`	A getter or a `protected` method
Accessing a `protected` member from another package through a supertype reference	`name has protected access in Animal`	Access through the subclass's own type
Narrowing access when overriding	`attempting to assign weaker access privileges; was public`	The same or wider access
Changing the return type	`return type String t not compatible with Animal`	The same type or a subtype
Adding a checked exception	`overridden method does not throw Exception`	Catching the exception inside the method
`@Override` with a wrong signature: `Draw()`, `equals(Point)`, a `private` method	`method does not override or implement a method from a supertype`	Exactly the same name and parameter list
Overriding a `final` method	`overridden method t final`	A new method with another name
Deriving from a `final` class	`cannot inherit from final Fixed`	Composition
Deriving from two classes	`'{' expected`	Composition or an interface
Calling a subclass method through a supertype variable	`cannot find symbol … method isWinter()`	`instanceof` and a downcast

Table 4.3.17: Common compile errors related to inheritance.

Mistake	Result	Fix
Comparing objects with `==`	`false` for two objects with the same content	`equals`
`equals` without `hashCode`	The same point is added to a `HashSet` twice: `2 false`	`Objects.hash(x, y)`
Writing `equals(Point)` (without `@Override`)	`contains` cannot find an equal object	`equals(Object)` and `@Override`
A typo without `@Override` (`Draw`)	The superclass version runs silently	`@Override` on every override

Mistake	Result	Fix
Thinking one has overridden a `private` method	The superclass code calls its own `private` version	`@Override` catches this misunderstanding
An unchecked downcast	`ClassCastException`	`instanceof` first
Calling an overridable method from a superclass constructor	The fields of the subclass are still at default values: `b=0 s=null`	Call only `private` / `final` methods from a constructor
Overriding methods that call each other together	The counter goes up twice: `6 3`	Composition and forwarding
Inheritance just to share code (`MyStack extends ArrayList`)	Unwanted `public` methods come along, insertion in the middle is possible	Composition
A subclass that breaks the promise (`Square extends Rectangle`)	Code written for the superclass gives a wrong result: 16	A common superclass or immutable classes

Table 4.3.18: Common run-time and logic errors related to inheritance.

Most of these mistakes are prevented by two habits. The first is writing `@Override` on every override; it turns three of the first five mistakes in the second table into compile errors. The second is applying the is-a test to the behavior of the class before setting up inheritance. If the test is negative or doubtful, composition is chosen.

4.3.9 Summary

- Inheritance is creating a new class by taking the members of an existing class and adding new ones. It is written `class Child extends Parent`; only one class follows `extends`. A class that does not write `extends` derives from `Object`.
- A subclass uses the `public` and `protected` members of its superclass as its own. `private` fields exist in the object but cannot be accessed directly. Constructors are not inherited.
- `new` creates a single object; the superclass part is inside this object. `super` is not a separate object but a way of reaching the same object through the superclass's code.
- `super(...)` calls the superclass constructor and in Java 21 is the first statement of the constructor. If it is not written, the compiler inserts `super();`. `super.m()` calls the superclass version and is bound at compile time.
- Constructor calls go from the bottom up, and the bodies complete in the order `Object` → … → `Parent` → `Child`. The order of object construction is: superclass fields, superclass constructor, subclass fields, subclass constructor. `static` blocks run once, when the class is first used, from the top down.
- Overriding is rewriting a method with the same signature in the subclass. The actual class of the object decides which version runs. Overloading is a different parameter list and is chosen at compile time.

- An overriding method cannot narrow the promise of the superclass. The return type is the same or a subtype, the access is the same or wider, and checked exceptions are the same or narrower. `@Override` has the compiler check these rules.
- `private`, `static` and `final` methods are not overridden. A method with the same name as a `private` one is a new method; a `static` method is only hidden.
- `protected` = class + same package + subclasses. A subclass in another package reaches a `protected` member only through a reference of its own type. Fields stay `private`, and `protected` methods are opened to subclasses.
- A `final` method cannot be overridden, and no subclass can be derived from a `final` class (`String`, `Math`, `Integer`).
- Upcasting happens by itself and narrows the view of the object: the type of the variable determines which methods can be called. A downcast is written explicitly and is first checked with `instanceof`.
- is-a is inheritance, has-a is composition. Java does not allow multiple inheritance of classes (the diamond problem); composition and interfaces are used instead.
- Inheritance brings tight coupling: the fragile base class problem and violations of the substitution principle are its consequences. If the only goal is to share code, composition and forwarding are chosen.
- `Object` is the ancestor of every class. `toString` prints the object readably. `equals` compares contents and obeys the contract. If `equals` is overridden, `hashCode` is overridden too; equal objects have the same hash code.
- `==` compares references, `equals` compares contents. `HashSet` first chooses a bucket with `hashCode`, then asks `equals`; with a good `hashCode`, operations take O(1) on average.

4.3.10 Key terms

inheritance superclass / subclass `super` keyword constructor chaining default constructor initializer block method overriding method overloading method signature annotation covariant return type access modifier upcasting / downcasting dynamic binding is-a / has-a relationship composition multiple inheritance fragile base class forwarding Liskov substitution principle `Object` class hash code

---

4.3.11 Test Yourself

1. Given `class X { X() { print("X "); } }`, `class Y extends X { Y() { this(1); print("Y "); } Y(int a) { print("Y" + a + " "); } }` and `class Z extends Y { Z() { super(5);`

`print("Z "); } }`. What does `new Z()` print?

Answer

The first line of `Z()` is `super(5)`, which calls `Y(int)` directly; `Y()` never runs. `Y(int)` has an implicit `super()`: first `X()` runs and prints `X`. Then the body of `Y(int)` prints `Y5`, and last the body of `Z()` prints `Z`. Result: `X Y5 Z`. The text `Y` is not printed, because the constructor containing `this(1)` was not called.

2. Given `class P { int f() { return 1; } int g() { return f() * 10; } }` and `class Q extends P { int f() { return 2; } int g() { return super.g() + 1; } }`. What is `p.g()` for `P p = new Q();`?

Answer

The object is a `Q`; `Q.g()` runs and calls `P.g()` with `super.g()`. The `f()` inside `P.g()` is `this.f()`, and `this` is still a `Q`: `Q.f()` returns 2. `P.g()` = 2 · 10 = 20, `Q.g()` = 20 + 1 = **21**. `super` decides only which version of `g` is called; the calls inside that version are still chosen by the object. `new P().g()` gives 10.

3. Which pairs are is-a and which are has-a relationships? Transistor–Semiconductor device, Circuit–Resistor, Phone–Battery, Vacuum cleaner–Household appliance.

Answer

is-a: Transistor–Semiconductor device ("every transistor is a semiconductor device") and Vacuum cleaner–Household appliance. **has-a**: Circuit–Resistor (a circuit has resistors; the resistors are kept in a list field) and Phone–Battery (a phone has a battery; the battery can be replaced, which fits the property of composition that the field can be changed later).

4. Given `class U { U() { System.out.println("U(): " + name()); } String name() { return "U"; } }` and `class V extends U { String s = "V"; String name() { return s; } }`. What does `new V()` print?

Answer

`U(): null`. The implicit `super()` of `V()` runs `U()`. The `name()` inside `U()` is overridden, and since the object is a `V`, it goes to `V.name()`. But the subclass's field initializers are assigned after `super()` returns. At that moment `s` is still `null`.

5. The `equals` of class `PersonI` uses `instanceof PersonI` instead of `getClass()` and compares name and age. `StudentI extends PersonI` adds no new field and does not override `equals`. What do `new PersonI("Ali", 20).equals(new StudentI("Ali", 20))` and the call in the reverse

order give?

Answer

Both are `true`. A `StudentI` is a `PersonI`; `instanceof` is true in both directions and the fields are equal. Symmetry is preserved, because both calls run the same `equals`. With `getClass()`, both calls would give `false`.

6. `Parent` has `protected Object copy()`. Inside `Child extends Parent`, does `@Override public Child copy()` compile? Does `@Override Child copy()` (no modifier) compile?

Answer

The first **compiles**. `Child` is a subtype of `Object` (covariant return) and the access widens from `protected` to `public`. The second **does not compile**: writing no modifier means package access, which is narrower than `protected`. javac says `attempting to assign weaker access privileges; was protected`.

7. Do `class A { final void f() { … } }` and `class B extends A { void f(int x) { … } }` compile?

Answer

They compile. The parameter list of `f(int)` is different; it does not override `A.f()`, it overloads it. `final` forbids only overriding. A `B` object has both methods: `b.f()` runs the version of `A`, `b.f(3)` the one of `B`.

8. `animals.Animal` has `protected String name`; `animals.Dog extends Animal`. In class `indoor.Cat extends Animal`, in another package, we write `String f(Dog k) { return k.name; }`. Does it compile?

Answer

It does not compile: `name has protected access in Animal`. A subclass in another package can reach a `protected` member only through a reference of its own type (`Cat` or its subtypes). `k` is a `Dog`; a cat is not allowed to touch the internal data of a dog. Had `Cat` been in package `animals`, it would have compiled.

9. In a class `Point` whose hash code is `Objects.hash(x, y)`, (3, 4) and then (1, 2) are added to an

empty `HashSet`. Which buckets do the two points fall into? What is `size()`?

Answer

(3, 4): 31 · 34 + 4 = 1058, 1058 % 16 = 2. (1, 2): 31 · 32 + 2 = 994, 994 % 16 = 2. The two points fall into **the same bucket**. On the second insertion the set asks `equals` against the (3, 4) in the bucket, gets `false`, and adds (1, 2) as well. Both `add` calls return `true`; `size()` = **2**. The same bucket does not mean equal objects.

4.3.12 Further reading

- Oracle, *The Java Tutorials*, the "Inheritance" lesson: covers subclasses, overriding and hiding, `super`, the `Object` class and `final` with short examples. It is Java's official, free starting point.
- J. Bloch, *Effective Java* (3rd ed.), Items 10–11 and 18–19: explains how to write `equals` and `hashCode`. It also discusses why composition is preferred over inheritance and the rules for designing for inheritance. The fragile base class example comes from this book too.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals* (12th ed.), Chapter 5 "Inheritance": discusses in detail the choice between `getClass()` and `instanceof` in `equals`, and the symmetry problem.
- OpenJDK, JEP 513 "Flexible Constructor Bodies": describes the change in Java 25 that allows statements before `super(...)`, and its rationale. It also explains why the constructor rules were introduced.

4.3.13 References

1. Oracle, *The Java Tutorials*: Inheritance; Overriding and Hiding Methods; Using the Keyword super; Object as a Superclass.
2. J. Gosling et al., *The Java Language Specification, Java SE 21 Edition*, Oracle, 2023 (8.1.4 Superclasses and Subclasses, 8.4.8 Inheritance, Overriding, and Hiding).
3. P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson (Chapter 9: Object-Oriented Programming: Inheritance).
4. K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly, 2022 (Chapter 7: Inheritance and Polymorphism).
5. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson, 2022 (Chapter 5: Inheritance).
6. J. Bloch, *Effective Java*, 3rd ed., Addison-Wesley, 2018 (Items 10–11: equals and hashCode; Item 18: favor composition over inheritance).
7. OpenJDK, *JEP 513: Flexible Constructor Bodies*, JDK 25, 2025. https://openjdk.org/jeps/513

PART II · CHAPTER 4.4

Polymorphism

In a zoo, the keeper says “everybody make a sound”; the dog barks and the cat meows. The keeper does not give each animal a separate command; every animal answers the same command in its own way. In object-oriented programming this idea is called polymorphism. The code is written once against the superclass type; at run time each object answers the call with its own version. These notes first establish the difference between the reference type and the object type: the compiler looks at one, the Java virtual machine (JVM) at the other. Upcasting and downcasting, `ClassCastException` and the `instanceof` test follow. Dynamic binding, method lookup and the method table are at the center of the notes. Why fields, `static` methods and overloading are resolved at compile time is examined separately. The notes end with a shape hierarchy, polymorphic parameters, an anti-tank game, and the symmetry trap of the `equals` method.

LEARNING OUTCOMES

Concept
Define polymorphism; tell compile-time polymorphism from run-time polymorphism.

Reference rule
Explain why a subclass object can be assigned to a superclass reference and why the reverse is a compile error.

Type conversion
Write upcasts and downcasts; prevent `ClassCastException` with `instanceof`; recognize the `ArrayStoreException` that array covariance causes.

Dynamic binding
Trace step by step how the method to call is chosen by the actual class of the object; show method lookup in the hierarchy and in the method table.

Overloading and overriding
Compare overloading and overriding; know which one the compiler chooses and which one the JVM chooses; apply the three phases of overload resolution.

Design
Write general code against the superclass type, using the shape array and the anti-tank game as examples; recognize the symmetry trap of the `equals` method.

4.4.1 The concept of polymorphism

The four basic principles of object-oriented programming are encapsulation, inheritance, polymorphism and abstraction. Encapsulation gathers data and the methods that work on it in one class. Inheritance lets a class extend another class. Polymorphism is built on these two: without a class hierarchy there is no run-time polymorphism either. That is why the basic concepts of inheritance are used here too: superclass, subclass, `extends`, overriding and `@Override`.

4.4.1.1 What is polymorphism?

Definition 4.4.1

Polymorphism means that the same expression leads to different behavior in different objects. The word comes from the Greek *poly* (many) and *morph* (form). In Java, a reference can refer to an object of its own type or of one of its subclasses. Which version of an overridden method called through this reference runs is determined by the actual class of the object it refers to.

The core of the definition can be seen in a single line. The call `animal.makeSound()` prints `Woof woof!` if `animal` refers to a dog and `Meow!` if it refers to a cat. The line is the same; the behavior changes with the object. The code is written once with the superclass type; each object answers the call with its own version.

Everyday life has many examples of this idea. The same “on” button on a remote control turns on the television, the air conditioner and the projector; each device does the job in its own way. With a class B driving licence you can drive any make of car. The driver uses only the “car” interface: steering wheel, pedals, gears. The details of the engine belong to the make. The keeper at the zoo also gives a single command and every animal makes its own sound. In all three examples the caller knows a common interface; the object completes the details itself.

```
Animal h1 = new Dog();
Animal h2 = new Cat();

h1.makeSound(); // Woof woof!
h2.makeSound(); // Meow!
```

In these four lines both variables have type `Animal`. `h1` refers to a `Dog` and `h2` to a `Cat`. The compiler compiles both calls in the same way, as a call to the method `makeSound()` of class `Animal`. When the program runs, the JVM looks at the object at each call and runs the version of `Dog` for the dog and that of `Cat` for the cat. The output is as written in the comments of the code: first `Woof woof!`, then `Meow!`.

4.4.1.2 Two kinds of polymorphism

In Java, "same name, different behavior" is obtained by two different mechanisms. The first is overloading: several methods with the same name but different parameters are written in the same class. The compiler decides which one to call. The second is overriding: a subclass rewrites a method it took from the superclass with the same signature. The JVM decides which version runs while the program is running.

Definition 4.4.2

Compile-time polymorphism (static polymorphism) means that the compiler chooses among overloaded methods by looking at the number and declared types of the arguments. **Run-time polymorphism** (dynamic polymorphism) means that the choice among overridden methods is made while the program runs, by looking at the actual class of the object. This second choice is called **dynamic binding**.

The two kinds can be seen side by side in the two short examples below. In the example on the left there are two `add` methods in a single class; if the arguments are integers the `int` version is chosen, if they are decimal the `double` version. This choice is made while the program is compiled and is written into the bytecode. In the example on the right the choice depends on the object and becomes known only while the program runs.

```
int add(int a, int b) { ... }
double add(double a, double b) { ... }

add(5, 3);     // int version
add(3.5, 2.7); // double version
```

	Compile time (static)	Run time (dynamic)
Mechanism	Overloading	Overriding + dynamic binding
Who chooses	The compiler (`javac`)	The JVM, while the program runs
What is looked at?	The number and declared types of the arguments	The actual class of the object
Is inheritance needed?	No, it works within a single class	Yes, a class hierarchy is needed
In short	Same name, different parameters	Same signature, different classes

Table 4.4.1: Comparison of the two kinds of polymorphism.

In everyday conversation, “polymorphism” usually means run-time polymorphism. Most of these notes are about it. Overloading is also examined separately, because the two mechanisms can work together in the same call, and this is where most of the confusion arises.

4.4.1.3 The example class hierarchy

Most of these notes use the same small hierarchy. Class `Animal` has three methods: `makeSound()`, `run()` and `sleep()`. `Dog` and `Cat` derive from this class. Like every class, `Animal` derives from `Object`, the ancestor of all classes in Java. Figure 4.4.1 shows this structure as a UML class diagram.

```
class Animal {
  public void makeSound() { System.out.println("Animal sound"); }
  public void run()      { System.out.println("Animal runs"); }
  public void sleep()      { System.out.println("Animal sleeps"); }
}
class Dog extends Animal {
  @Override public void makeSound() { System.out.println("Woof woof!"); }
  @Override public void run() { System.out.println("Dog runs"); }
  public void bark() { System.out.println("Dog barks"); }
}
class Cat extends Animal {
  @Override public void makeSound() { System.out.println("Meow!"); }
  public void scratch() { System.out.println("Cat scratches"); }
}
```

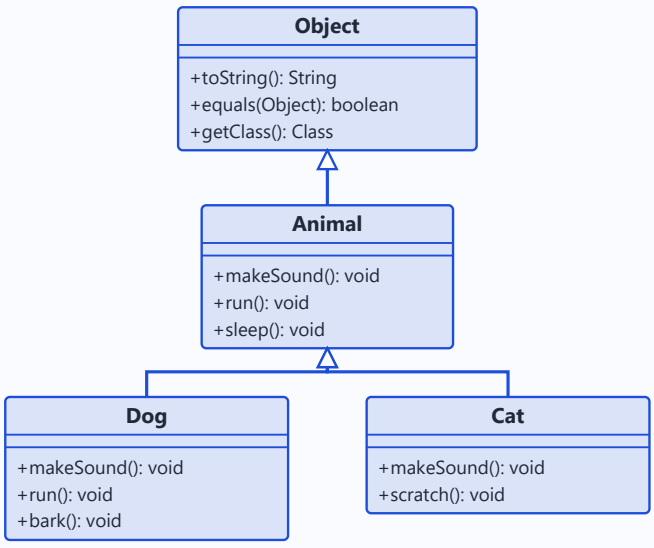


Figure 4.4.1: The example class hierarchy. `Dog` and `Cat` derive from `Animal`; `Animal` derives from `Object`. The arrows point from the subclass to the superclass.

The two subclasses relate to the superclass differently. `Dog` overrides `makeSound()` and `run()` and adds its own `bark()` method. `Cat` overrides only `makeSound()` and adds `scratch()`. Neither subclass overrides `sleep()`; both take it from `Animal` as it is. `run()` has changed only in the dog; the cat uses the version of `Animal`.

Each of these details corresponds to a rule. `bark()` exists only in `Dog`, so it cannot be called through a reference of type `Animal`. Since `sleep()` is not written in the subclasses, the method lookup climbs up to `Animal`. The `@Override` annotation tells the compiler that "this method overrides a method in the superclass". If the name or a parameter is misspelled, the compiler reports an error.

The is-a relationship of inheritance is the basis of all the rules here. A dog and a cat are both animals. That is why both can be held in a variable of type `Animal`. The reverse is not true: not every animal is a dog.

4.4.2 References and type compatibility

To understand polymorphism, first remember that variables in Java hold not the object but a reference to it. The object lives on the heap. The variable sits in a frame on the stack and points to the location of the object on the heap. Many references can point to one object, and the types of these references can differ.

4.4.2.1 One object, three references

The code below creates a single `Dog` object and refers to it with references of three different types.

```
// reference to a Dog object
Dog dog = new Dog();
// an Animal reference can point to a Dog
Animal animal = dog;
// so can an Object reference
Object obj = animal;
System.out.println(dog == animal);
System.out.println(animal == obj);
System.out.println(obj.getClass());
```

The program prints:

```
true
true
class Dog
```

Let us follow the code line by line. The `new Dog()` in the first line creates an object on the heap. The reference `dog` points to this object; the type of the reference is `Dog`. The second assignment does not create a new object. `animal` is a second reference that points to the same object. The assignment is valid, because a dog is an animal. In the third assignment, `obj` of type `Object` also points to the same object. Since every class derives from `Object`, this is valid too.

The next two lines use the `==` operator. For reference types, `==` compares not the contents of the objects but whether the references point to the same object. Since all three references point to the same object, both comparisons give `true`. The `getClass()` in the last line returns the run-time class of the object. Even though the type of the reference is `Object`, the class of the object has not changed: `class Dog`.

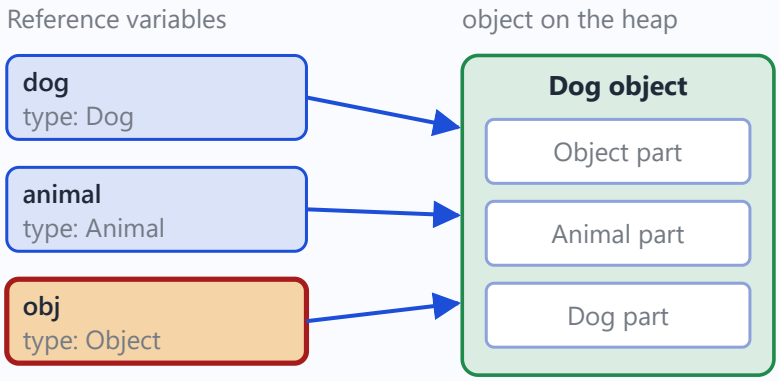


Figure 4.4.2: Three references, one object. `dog`, `animal` and `obj` point to the same `Dog` object on the heap. Inside the object are the parts that come from `Object`, `Animal` and `Dog`.

Figure 4.4.2 also shows the internal structure of the object. A `Dog` object carries the parts from every class in the inheritance chain: the `Object` part, the `Animal` part and the `Dog` part. That is why the object can be referred to by a reference of any type in the chain. Each reference points to the whole object, but sees the object through the window of its own type.

4.4.2.2 Reference type and object type

The previous example shows that every reference variable has two separate types. One is the type written in the variable's declaration. The other is the class of the object the variable currently refers to.

Definition 4.4.3

The **declared type** of a reference variable is the type written in its declaration and does not change during the variable's lifetime. Its **actual type** is the class of the object the variable currently refers to, and it can change with assignment. The declared type is also called the reference type, and the actual type the object type.

```
Animal h = new Dog();
// declared type: Animal
// actual type:   Dog
```

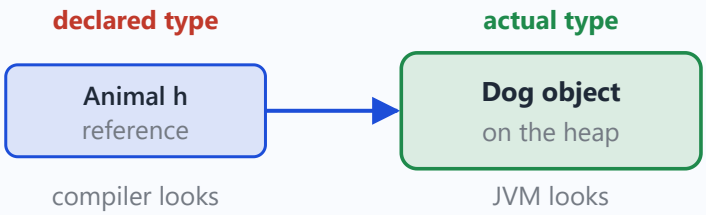


Figure 4.4.3: The two types of the line `Animal h = new Dog();`. The declared type is `Animal` and the compiler looks at it. The actual type is `Dog` and the JVM looks at it.

The two types have different functions, and this distinction is the key to the rest of the notes. The compiler checks the code before the program runs. At that moment no object exists; the compiler knows only the declarations of the variables. So all of the compiler's decisions rest on the declared type. The JVM, while the program runs, meets the real object and knows which class it belongs to. Table 4.4.2 summarizes three basic questions and who answers them.

Question	Who decides?
Can this method be called?	The compiler, by looking at the reference type.
Which version runs?	The JVM, by looking at the object type.
Which assignments are valid?	The compiler, by looking at the reference type.

Table 4.4.2: Reference type and object type: who decides which question?

The result of the first question can be seen in the code below. The object really is a dog, but the type of the variable is `Animal`.

```
Animal animal = new Dog();
animal.bark(); // compile error
```

```
error: cannot find symbol
  symbol:   method bark()
  location: variable animal of type Animal
```

When the compiler reaches the line `animal.bark()`, it knows only this: `animal` is a variable of type `Animal`. There is no method called `bark()` in class `Animal` or in its superclass `Object`. The compiler cannot know which object the variable refers to. When the program reaches that line, the

variable may be pointing at a cat. That is why it does not allow the call. The line `location: variable animal of type Animal` in the message also says plainly that the decision was made by the type of the variable.

The remote control analogy describes this well. The reference type is a remote control; only the buttons the remote has can be used. Even if the object is a dog, the `Animal` remote has no `bark()` button. What the device does when a button is pressed is determined by the device, that is, the object. The `makeSound()` button is on the `Animal` remote; if the object is a dog, a bark is heard when it is pressed, if a cat, a meow.

Common mistake

Thinking "the object is a dog, so `bark()` can be called". Whether a call compiles is decided not by the object but by the declared type of the reference. If `bark()` is to be called, the reference is either declared as `Dog` from the start or converted to `Dog` by a downcast.

4.4.2.3 The reference rule

The validity of assignments is also checked against the declared types. Java's rule on this is short and comes from the is-a relationship of inheritance.

Definition 4.4.4

The reference rule: An object of a subclass can be assigned to a reference of the superclass type. An object of the superclass cannot be assigned to a reference of a subclass type. In short, assignment from a subclass reference to a superclass reference is automatic; the reverse needs an explicit cast.

```
// Dog is a subtype of Animal
Animal animal = new Dog();
Object obj = new Cat();
```

These two assignments are valid. Every dog is an animal: a dog can do everything expected of an animal, that is, `makeSound()`, `run()` and `sleep()`. Every cat is an `Object` too. A reference of a supertype cannot ask for anything the object of the subtype cannot do. That is why the assignment is safe.

```
// Animal is Dog's superclass
Dog dog = new Animal();
```

```
error: incompatible types:
  Animal cannot be converted to Dog
```

This assignment, however, does not compile. Through the variable `dog`, `dog.bark()` could be called later. A plain `Animal` object does not know `bark()`. Had the compiler allowed such an assignment, the program would try to call a nonexistent method at run time. Java prevents this kind of error up front, at compile time.



Figure 4.4.4: The reference rule. An `Animal` reference can point to a `Dog` object; a `Dog` reference cannot point to an `Animal` object.

The most surprising consequence of the rule can be seen in these two lines:

```
Animal h = new Dog();
Dog k = h;   // compile error
```

```
error: incompatible types: Animal cannot be converted to Dog
```

Here the object that `h` points to really is a dog. Even so, the second line does not compile, and the same message appears. The compiler sees not the object `h` points to but only its declared type (`Animal`). It does not read the previous line and conclude "so this is a dog"; it could not do that in every case anyway. The solution is an explicit cast: `Dog k = (Dog) h;`. With this notation the programmer tells the compiler "I am sure this reference points to a dog". Whether the cast is correct is checked at run time.

Example 4.4.1

`Dog` and `Cat` derive from `Animal`. Which of the following four assignments compile?

(a) `Animal h = new Cat();` — Compiles. A cat is an animal; a subclass object is assigned to a superclass reference.

(b) `Object o = new Dog();` — Compiles. Every class is a subclass of `Object`.

(c) `Cat c = new Dog();` — Does not compile: `incompatible types: Dog cannot be converted to Cat`. `Dog` and `Cat` are sibling classes; there is no is-a relationship between them.

(d) `Dog k = new Animal();` — Does not compile: `incompatible types: Animal cannot be converted to Dog`. A superclass object cannot be assigned to a subclass reference.

4.4.3 Type conversion

Conversion is a familiar operation with primitive types. Going from `int` to `double` happens by itself; `(int) 3.7` is an explicit cast and changes the value. With reference types there are two similar directions, but with an important difference: a reference conversion never changes the object. Only the type of the reference looking at the object changes.

4.4.3.1 Upcasting and downcasting

Definition 4.4.5

Upcasting is converting a reference of a subclass type to a superclass type. It is automatic and always safe. **Downcasting** is converting a reference of a superclass type to a subclass type. It is done explicitly by writing the target type in parentheses: `(Dog) h`. Its success depends on the object the reference points to.

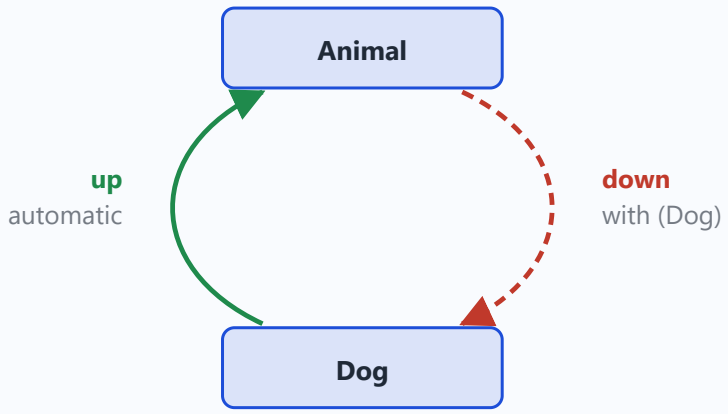


Figure 4.4.5: Upcasting from `Dog` to `Animal` happens by itself. Downcasting from `Animal` to `Dog` is done with an explicit cast, by writing `(Dog)`.

```
Dog dog = new Dog();
Animal h = dog;    // up: automatic
Animal h1 = (Animal) dog; // unnecessary
```

On the second line the upcast is done by itself; it is the reference rule itself. The `(Animal)` on the third line is valid too, but unnecessary. The compiler does this conversion itself anyway.

```
Animal h = new Dog();
Dog k = (Dog) h; // down: explicit
k.bark();           // Dog barks
```

Here we go in the opposite direction. The declared type of `h` is `Animal`; the type of the expression `(Dog) h` is `Dog`. This expression can be assigned to `k` of type `Dog`, and `k.bark()` compiles. While the program runs, the JVM checks whether the object `h` points to really is a `Dog`. Since the object is a dog, the cast succeeds and `Dog barks` is printed.

	Upcasting	Downcasting
Direction	Sub → super	Super → sub
Notation	Automatic (implicit)	Explicit: `(Dog) h`
Safe?	Always	Depends on the object
Other name	Widening conversion	Narrowing conversion

Table 4.4.3: Comparison of upcasting and downcasting.

The names “widening” and “narrowing” in the table are familiar from primitive types. The Java Language Specification uses these names for reference conversions too. An upcast counts as “widening” even though it narrows the set of methods the reference can see. The name comes from the widening of the set of objects the type covers: the type `Animal` covers all dogs, cats and other animals.

Common mistake

Thinking “a cast changes the object”. Writing `(Animal) dog` does not turn the dog into a plain animal. The call `((Animal) dog).makeSound()` still prints `Woof woof!`, because the object decides which version runs. A cast changes only the type in which the compiler sees that expression.

4.4.3.2 Downcasting and the precedence trap

The purpose of downcasting is to call methods that are not in the superclass and belong only to the subclass. A superclass reference can be assigned to a subclass variable only through this cast. The cast is written in two forms: its result is assigned to a variable, or it is used directly on the same line.

```
Animal h2 = new Dog();
Dog k = (Dog) h2;
k.bark();
((Dog) h2).bark(); // in one line
```

```
Dog barks
Dog barks
```

The program prints `Dog barks` twice. In the first form the result of the cast is assigned to `k` and `k.bark()` is called. In the second, the expression `((Dog) h2)` produces a `Dog` reference and `.bark()` is applied directly to it. The outer pair of parentheses is mandatory. The reason is operator precedence in Java.

Definition 4.4.6

The precedence rule: Member access and method call (`.`) are applied before the cast operator. The expression `(Dog) h2.bark()` is read as `(Dog) (h2.bark())`: the cast is meant to apply to the result of the method call.

```
(Dog) h2.bark();   // wrong
```

```
error: not a statement
```

When the compiler parses this line, it sees first the call `h2.bark()`, then the conversion of its result to `Dog`. Such a cast expression cannot be a statement on its own; the compiler says `not a statement`. Even if the line is fixed in some other way, a second problem remains: the declared type of `h2` is `Animal`, and `Animal` has no `bark()`.

When the expression is used on the right of an assignment, the second problem shows directly:

```
String s = (Dog) h2.name();
// h2.name() first → not in Animal
```

```
error: cannot find symbol
  symbol:   method name()
  location: variable h2 of type Animal
```

Here too the compiler first tries to resolve `h2.name()`. It cannot find `name()` in class `Animal` and reports an error. For this line, javac of JDK 21 also gives a second message: `incompatible types: Dog cannot be converted to String`. The result of the cast is a `Dog` and cannot be assigned to a `String` variable. The correct form is always the same: the cast is put in parentheses and the dot is written outside the parentheses: `((Dog) h2).bark();`.

4.4.3.3 `ClassCastException`

A downcast is a promise given to the compiler. The compiler cannot know whether the promise is true and accepts it. If the promise is not kept, the error appears while the program runs.

```
Animal h4 = new Cat();
Dog k4 = (Dog) h4; // compiles!
k4.bark();
```

```
Exception in thread "main" java.lang.ClassCastException:
class Cat cannot be cast to class Dog
```

The second line compiles. From the compiler's point of view there is no problem in it: a reference of type `Animal` may well be pointing at a `Dog`. When the program runs, the JVM checks the actual class of the object at the moment of the cast. The object is a `Cat`, and a cat is not a dog. In this case the JVM throws `ClassCastException`; since nobody catches it, the program stops. The third line never runs.

Definition 4.4.7

`ClassCastException` is the run-time exception the JVM throws when a reference is explicitly cast to a type that is incompatible with the class of the object it points to. The cast `(T) r` succeeds only if the object `r` points to is an instance of `T` or of a subclass of `T`. If `r` is `null`, the cast always succeeds and the result is `null` again.

In JDK 21 the whole message is a little longer: `class Cat cannot be cast to class Dog (Cat and Dog are in unnamed module of loader 'app')`. The part in parentheses says from which module and class loader the classes come; it is not important in this course. The first part of the message tells everything: which class was asked to be cast to which class.

The same exception appears when a plain superclass object is cast to a subclass:

```
Dog k5 = (Dog) new Animal(); // compiles!
```

```
java.lang.ClassCastException:
class Animal cannot be cast to class Dog
```

This line compiles too, because the type of the expression is `Animal` and a cast from `Animal` to `Dog` is possible in the compiler's eyes. But the object that was created is not a dog and never will be. The general conclusion is this: an object of the superclass itself can never be assigned to a subclass reference. The cast succeeds only if the object really is of that subclass.

There is also a kind of cast the compiler does not accept at all. If a cast is written between two types that have no inheritance relationship, the compiler knows it cannot be right for any object:

```
Dog dog = new Dog();
Cat c = (Cat) dog;   // compile error
```

```
error: incompatible types: Dog cannot be converted to Cat
```

`Dog` and `Cat` are sibling classes. An object cannot be both a dog and a cat at the same time; in Java a class has only one superclass. So the compiler rejects the cast at compile time. For classes the rule is briefly this: a cast does not compile if neither the source type nor the target type is a subtype of the other; if one is a subtype, it compiles and is checked at run time.

Example 4.4.2

Given `Animal h4 = new Cat();`. What are the decisions of the compiler and the JVM for the following three casts?

(a) `(Cat) h4`: `Cat` is a subtype of `Animal`; it compiles. The object is a cat; the JVM approves. Succeeds.

(b) `(Dog) h4`: `Dog` is a subtype of `Animal`; it compiles. The object is a cat; the JVM throws `ClassCastException`.

(c) `(Object) h4`: it is an upcast, compiles and always succeeds. The object does not change; the type of the reference becomes `Object`.

4.4.3.4 Code trace: a faulty downcast

The program below contains one successful and one failing downcast. The trace is easier to read if the lines are numbered from 1 to 7.

```
Animal h1 = new Dog();
Animal h2 = new Cat();
Dog k1 = (Dog) h1;
k1.bark();
Dog k2 = (Dog) h2;
k2.bark();
System.out.println("Done");
```

Line	Compiler	JVM (at run time)	State / output
1	Upcast, valid	`Dog` object #1 is created	`h1` → #1
2	Upcast, valid	`Cat` object #2 is created	`h2` → #2
3	`Animal` → `Dog` possible, compiles	Is #1 a `Dog`? Yes	`k1` → #1
4	`Dog` has `bark()`	`Dog.bark()`	`Dog barks`
5	`Animal` → `Dog` possible, compiles	Is #2 a `Dog`? `Cat` → `Animal` → `Object`: no	`ClassCastException`, nothing is assigned to `k2`
6	Compiles	Does not run	—
7	Compiles	Does not run	`Done` is not printed

Table 4.4.4: Trace of the faulty downcast program.

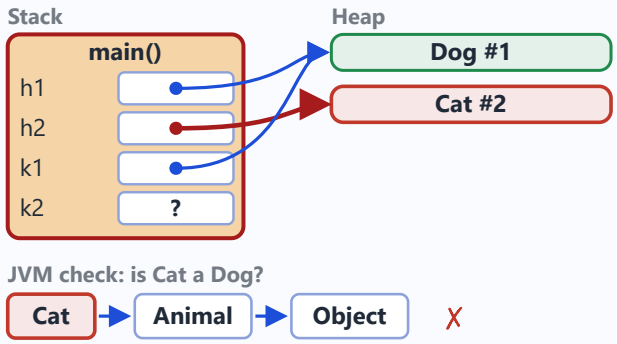


Figure 4.4.6: The JVM's check at line 5. In the chain going up from the class of the object `h2` points to (`Cat` → `Animal` → `Object`) there is no `Dog`; the cast fails and `k2` stays empty.

The most important row of the table is line 5. The compiler cannot tell this line apart from line 3; in both the declared type is `Animal` and the target type is `Dog`. The difference shows only at run time, when the object is looked at. The JVM's check is the chain walk seen in Figure 4.4.6: it starts at the class of the object and walks up the superclasses looking for `Dog`. If no `Dog` is found up to `Object`, the cast is rejected.

The output of the program is only the line `Dog barks`, followed by the exception message. `Done` is not printed, because when an exception is not caught, the `main` method ends on the spot.

4.4.3.5 Safe casting with `instanceof`

If it is not known in advance what class of object a reference points to, a test is made before the downcast. Java offers the `instanceof` operator for this.

Definition 4.4.8

The expression `reference instanceof Type` gives `true` if the object the reference points to is of `Type` or of one of its subclasses, and `false` otherwise. `null instanceof Type` is always `false`. If there is no inheritance relationship at all between the declared type of the reference and `Type`, the expression does not compile.

The loop below tests the type first and then casts, for each element of the array `animals = {new Dog(), new Cat(), new Animal()}`.

```
for (Animal x : animals) {
    if (x instanceof Dog) {
        Dog kk = (Dog) x;
        kk.bark();
    } else if (x instanceof Cat) {
        ((Cat) x).scratch();
    } else {
        System.out.println("No special behavior");
    }
}
```

```
Dog barks
Cat scratches
No special behavior
```

Since the test comes first and the cast second, `ClassCastException` cannot occur. If `x instanceof Dog` is true, the object `x` points to is certainly a dog and `(Dog) x` succeeds. The third element is a plain `Animal`; both tests come out false and the `else` branch runs.

The order of the tests matters too. `instanceof` includes subclasses. The test `x instanceof Animal` gives `true` for both a dog and a cat. That is why in a chain the lowest, most specific classes are tested first. If the order is reversed, the general branch catches every object and the specific branches never get their turn.

Pattern matching, which came with Java 16, shortens this idiom. If the test is true, the cast value is bound directly to a new variable:

```
if (x instanceof Dog kk) {
    kk.bark(); // the cast is automatic
}
```

The variable `kk` is defined only in the true branch of the `if` and has type `Dog`. The compiler writes the cast; the programmer need not write `(Dog) x` separately. Since the test and the cast are not on separate lines, a mistake like changing one and forgetting the other cannot be made either.

The way the tests work in each round is clearer if a version of the same loop with the elements in a different order is traced.

```
Animal[] arr = {new Cat(), new Animal(), new Dog()};
for (Animal x : arr) {
    if (x instanceof Dog) {
        ((Dog) x).bark();
    } else if (x instanceof Cat) {
        ((Cat) x).scratch();
    } else {
        System.out.println("No special behavior");
    }
}
```

Round	`x` → object	`instanceof Dog`	`instanceof Cat`	Branch that runs	Output
1	`Cat` #1	false	true	`((Cat) x).scratch()`	`Cat scratches`
2	`Animal` #2	false	false	`else`	`No special behavior`
3	`Dog` #3	true	— (not tested)	`((Dog) x).bark()`	`Dog barks`

Table 4.4.5: Trace of the `instanceof` chain. Array: `{Cat, Animal, Dog}`.

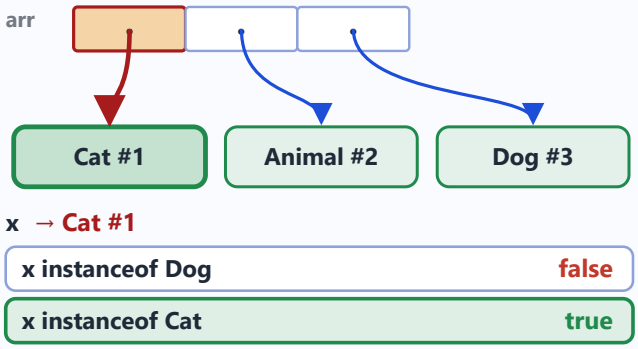


Figure 4.4.7: The first round. `x` points to a `Cat`; the `Dog` test is false and the `Cat` test is true. Since it was tested first, the cast `(Cat) x` is safe.

In the second round there is a plain `Animal` object. This object is an instance of no subclass; both tests come out false. In the `else` branch no cast is attempted, so there is no risk of an error. In the third round the first test is true; the next condition in the `else if` chain is not tested. In none of the three rounds does `ClassCastException` occur.

Long `instanceof` chains are often a design warning. When a new subclass is added (for example `Bird`), a new branch has to be added to the chain; if it is forgotten, the bird silently falls into the `else` branch. A better solution is to move the behavior into a method defined in the superclass and overridden in the subclasses.

4.4.3.6 Arrays and `ArrayStoreException`

The reference rule applies to arrays too. Since `Dog` is an `Animal`, a `Dog[]` counts as an `Animal[]` as well. This property of Java arrays is called covariance.

> **Definition 4.4.9**
>
> Arrays are **covariant**: if `A` is a subtype of `B`, then `A[]` is a subtype of `B[]`. So a `Dog[]` object can be assigned to an `Animal[]` reference. The array object, however, knows its own actual element type. Every element written into the array is checked against this type at run time; for an element that does not fit, `ArrayStoreException` is thrown.

```
Dog[] dogs = new Dog[2];
Animal[] animals = dogs;   // compiles
animals[0] = new Dog();
System.out.println("first ok");
animals[1] = new Cat();       // compiles, but...
System.out.println("never reached");
```

```
first ok
Exception in thread "main" java.lang.ArrayStoreException: Cat
```

On the second line an upcast is done and it compiles. The array object is really a `Dog[]`; `animals` is a second reference that only looks at it. On the third line a dog is written into the array; the JVM's check passes and `first ok` is printed. On the fifth line `animals[1] = new Cat();` is valid in the compiler's eyes: an `Animal` is written into an array of type `Animal[]`. But the JVM looks at the actual element type of the array at the moment of writing. A cat cannot be put into a `Dog[]` array; `ArrayStoreException: Cat` is thrown and the last line does not run.

This is a known gap in the type safety of arrays. The JVM catches an error the compiler could not catch with the check it makes at every write. Reading is safe: every element read from a `Dog[]` array is an `Animal`. The problem arises only when writing.

Generic collections close the same gap at compile time. `List<Dog>` is not a subtype of `List<Animal>`:

```
List<Animal> l = new ArrayList<Dog>();
```

```
error: incompatible types: ArrayList<Dog>
  cannot be converted to List<Animal>
```

Had the compiler allowed this assignment, a cat could be added to the list with `l.add(new Cat())`. Since generic type information is not kept at run time, the JVM could not catch it. That is why Java does not make generics covariant and turns the problem into a compile error up front.

Common mistake

Relying on array covariance: `Animal[] d = new Dog[2]; d[1] = new Cat();` compiles, but at run time gives `ArrayStoreException: Cat`. An array that will hold objects of different subclasses is created with the superclass type from the start: `new Animal[2]`.

4.4.4 Dynamic binding

Matching a method call with the method body that will run is called binding. This matching can be done at two times. If it is done during compilation, it is called static (early) binding; if it is done while the program runs, dynamic (late) binding. In Java, instance methods that can be overridden are bound dynamically, and this is what makes polymorphism possible.

4.4.4.1 The compiler asks “is there one?”, the JVM asks “which one?”

Definition 4.4.10

Dynamic binding is choosing the method version to call not by the declared type of the variable but by the actual class of the object, at run time. If the reference points to a subclass object and the subclass has overridden the method, the subclass's version runs.

```
Animal h = new Dog();
h.run();      // Dog's method
h.makeSound(); // Dog's method
h.sleep();      // not in Dog → Animal's
```

```
Dog runs
Woof woof!
Animal sleeps
```

All three calls are made through `h`, of type `Animal`. For `h.run()` and `h.makeSound()` the versions of `Dog` run, because the object is a dog and these two methods are overridden in `Dog`. For `h.sleep()` the version of `Animal` runs. `Dog` has not overridden `sleep()`; the version it received by inheritance is that of `Animal`.

Each call is resolved in two phases. Figure 4.4.8 shows these two phases for `h.run()`.



Figure 4.4.8: The two phases of the call `h.run()`. The compiler checks whether `Animal` has `run()`; the JVM looks at the class of the object and runs `Dog.run()`.

1. **Compile time.** The compiler looks at the declared type of `h` (`Animal`) and asks "does `Animal` have `run()`?" If it does, the call compiles; if not, a compile error is reported. At this phase the body to run is not chosen; only the validity of the call is settled.
2. **Run time.** The JVM looks at the object `h` points to and asks "which `run()` applies in the class of this object?" The actual class is `Dog`; `Dog.run()` runs.

The rule can be summed up briefly: the compiler asks "is there one?", the JVM asks "which one?" The compiler's approval guarantees that at least one version will be found at run time. Because whichever subclass object `h` points to, that class has either inherited or overridden the `run()` method of `Animal`.

4.4.4.2 Method lookup from the bottom up

How the JVM answers the "which one?" question is easiest to show on the class hierarchy. The search always starts at the bottom, at the actual class of the object. If the method is found there, the search ends. If not, it moves up to the class above. This walk can continue up to `Object`.

```
Animal h = new Dog();
h.run();
h.sleep();
System.out.println(h.toString());
// h.bark(); → compile error
```

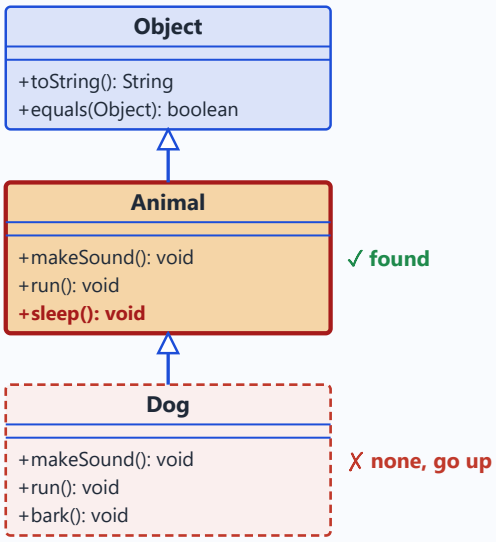


Figure 4.4.9: Method lookup for `h.sleep()`. The object is a `Dog`; `Dog` has no `sleep()`. The search moves one level up and ends at `Animal`.

Each of the four lines in the program shows a different situation:

- `h.run()`: the search starts at `Dog`. `run()` is there; the search ends and `Dog runs` is printed.
- `h.sleep()`: `Dog` has no `sleep()`. The search moves up and finds the method in `Animal`: `Animal sleeps` (Figure 4.4.9).
- `h.toString()`: it is in neither `Dog` nor `Animal`. The search climbs to the very top, to `Object`. The `toString()` of `Object` concatenates the class name, the `@` sign and a hexadecimal number (the hash code of the object); for example `Dog@13221655`. The number changes from run to run (Figure 4.4.10).
- `h.bark()`: does not compile. This time the compiler does the search, starting from the reference type `Animal`. `bark()` is not found in `Animal` or in `Object`. `Dog` is never looked at, because the compiler does not go down.

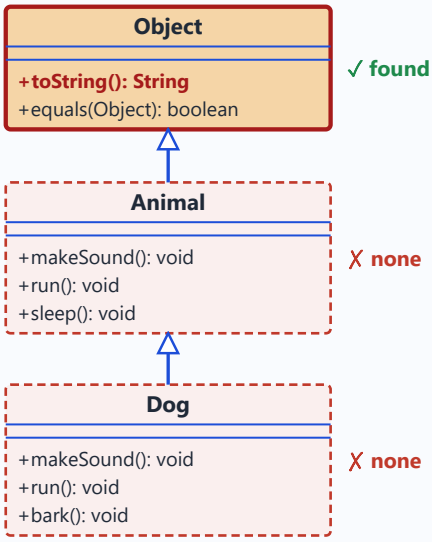


Figure 4.4.10: Method lookup for `h.toString()`. A method found neither in `Dog` nor in `Animal` is found in `Object` at the top of the chain.

The last item shows the difference between the two searches clearly. The compiler's search starts at the reference type and goes only up. The JVM's search starts at the class of the object and also goes only up. The point where the JVM starts is lower than or at the same place as the compiler's. That is why every method the compiler finds, the JVM finds too; perhaps an overridden version of it, lower down.

Example 4.4.3

For `Animal c = new Cat();`, which refers to a `Cat` object, where does the search end for the calls `c.makeSound()`, `c.run()` and `c.sleep()`?

`c.makeSound()`: the search starts at `Cat` and ends there, because `Cat` overrides this method: `Meow!`.

`c.run()`: `Cat` has no `run()`; the search ends at `Animal`: `Animal runs`. The same call on a dog prints `Dog runs`.

`c.sleep()`: ends at `Animal` again: `Animal sleeps`.

4.4.4.3 Code trace: a polymorphic array

The place where polymorphism is used most is an array or list of the superclass type. The elements of the array refer to objects of different subclasses; a single loop makes the same call on all of them.

```
Animal[] animals = new Animal[3];
animals[0] = new Dog();
animals[1] = new Cat();
animals[2] = new Animal();
for (int i = 0; i < animals.length; i++) {
    animals[i].makeSound();
}
```

The first line creates an array of three elements of type `Animal`. The elements are references, not objects, and at first all are `null`. The next three assignments bind a different object to each element. Element 0 points to a `Dog`, element 1 to a `Cat`; element 2 is a plain `Animal`. At every assignment an upcast is done automatically. The array now holds three different forms.

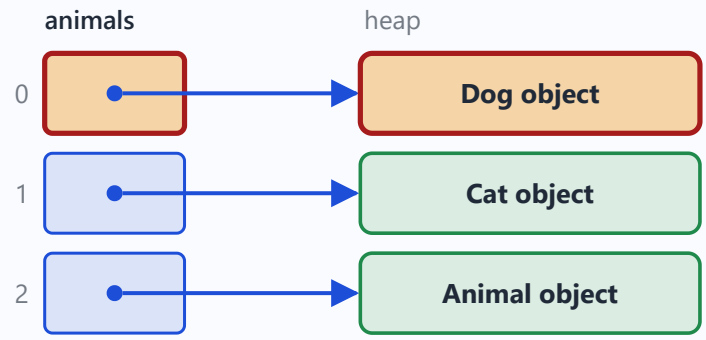


Figure 4.4.11: The first round of the loop. `animals[0]` points to a `Dog` object; the JVM runs `Dog.makeSound()`.

`i`	`i < animals.length`	`animals[i]` → object	Version that runs	Output
0	0 < 3 true	`Dog`	`Dog.makeSound()`	`Woof woof!`
1	1 < 3 true	`Cat`	`Cat.makeSound()`	`Meow!`
2	2 < 3 true	`Animal`	`Animal.makeSound()`	`Animal sound`
3	3 < 3 false	—	—	loop ends

Table 4.4.6: Trace of the polymorphic array loop.

The line `animals[i].makeSound()` in the loop body is the same in all three rounds. The compiler compiles this line once, as a call to `Animal.makeSound()`. The JVM looks at the class of the object in every round and chooses a different body. One line of code produces three different behaviors. If a `Bird` object were added to the array tomorrow, no change would be needed in the loop; the `makeSound()` version of `Bird` would run by itself.

Common mistake

Calling through the array before assigning objects to its elements. `new Animal[3]` does not create three animals; it creates three empty references. If `animals[0].makeSound()` is called without assignment, the program stops with `NullPointerException`.

4.4.4.4 Behind the scenes: the method table

Searching from the bottom of the hierarchy upward is a good model for understanding dynamic binding. JVMs, however, do not walk through the classes one by one at every call; that would be slow. In reality the lookup work is largely done once, while the class is loaded.

Definition 4.4.11

The **method table** (virtual method table) is a table the JVM uses in dynamic binding, kept once per class. Every method that can be overridden has a fixed sequence number in the table, its **slot**. The slot holds the address of the method body that runs on objects of that class.

Tables are built along inheritance like this. The table of a subclass starts as a copy of the superclass's table. The slots of overridden methods get the new body. New methods added by the subclass are appended to the end of the table. So the slot number of a method stays the same in all subclasses. Figure 4.4.12 shows the tables of the example hierarchy.

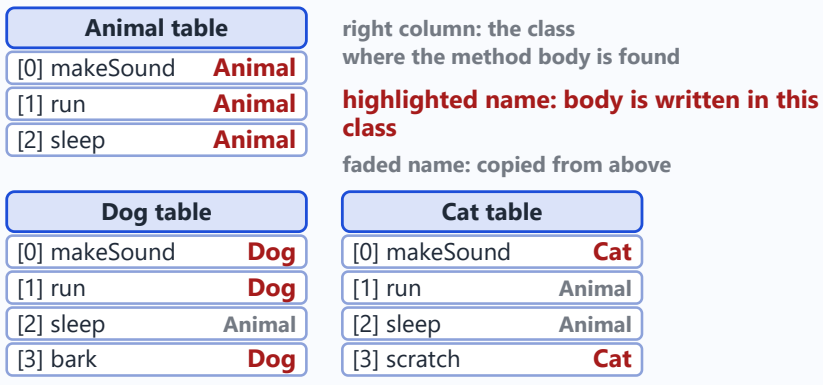


Figure 4.4.12: Method tables of the classes `Animal`, `Dog` and `Cat`. The right column is the class where the method body is found. A highlighted name means the body is written in that class; a faded name means it was copied from the superclass.

The tables in the figure are simplified. Real tables start with the overridable methods of `Object` (such as `toString`, `equals`, `hashCode`) and slot numbers vary with the JVM. The principle is the same. In the `Dog` table, slots 0 and 1 (`makeSound`, `run`) point to the bodies of `Dog`. Slot 2, `sleep`, is

copied from `Animal`. `bark` is appended at the end. In the `Cat` table only `makeSound` has changed, and `scratch` is appended at the end.

The bytecode the compiler produces uses this structure. The two lines below are taken from the output of the command `javap -c Main`. This command shows the bytecode of a compiled class in readable form.

```
47: invokevirtual #15  // Method Animal.run:()V
52: invokevirtual #18  // Method Animal.makeSound:()V
```

In both lines the compiler wrote only `Animal.run` and `Animal.makeSound`, because it did its check by the reference type. The names `Dog` and `Cat` do not appear in the bytecode. Which body will run is decided by the `invokevirtual` instruction. When this instruction runs, the JVM follows this path: reference → object → class of the object → the relevant slot in that class's table → body. The lookup is a single array access; there is no need to walk up the hierarchy.

Not all calls are made with `invokevirtual`. Constructors and `super.m()` calls use `invokespecial`, and `static` methods use `invokestatic`. With these instructions the body to run is known at compile time and the table is not consulted. Since Java 11, `private` instance methods are called with `invokevirtual`. But because they cannot be overridden, they have only one candidate and the JVM picks it directly.

```
Animal[] d = {new Dog(), new Cat()};
for (Animal h : d) {
    h.run();
    h.makeSound();
}
```

Call	Bytecode	Object	Slot in the table	Output
`h.run()`	`Animal.run`, slot 1	`Dog` #1	`Dog.run`	`Dog runs`
`h.makeSound()`	`Animal.makeSound`, slot 0	`Dog` #1	`Dog.makeSound`	`Woof woof!`
`h.run()`	`Animal.run`, slot 1	`Cat` #2	`Animal.run` (by inheritance)	`Animal runs`
`h.makeSound()`	`Animal.makeSound`, slot 0	`Cat` #2	`Cat.makeSound`	`Meow!`

Table 4.4.7: Trace of the calls through the method table. The bytecode is the same at every call; only the table of the object changes.

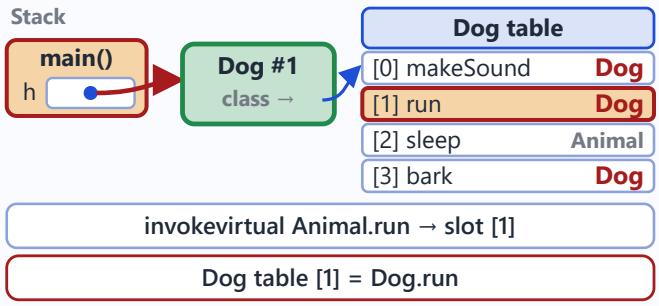


Figure 4.4.13: The first `h.run()` call. The object is a `Dog`; slot 1 of its table holds `Dog.run`.

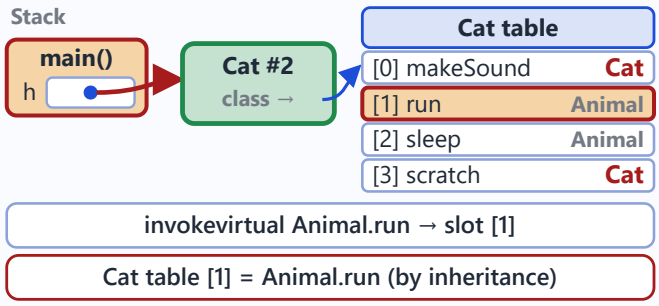


Figure 4.4.14: The same bytecode, another object. Since `Cat` does not override `run`, slot 1 of its table points to `Animal.run`.

The four-line output of the program is the last column of the table. In none of the four calls was an `if` or a type test made; the tables made the choice. Comparing Figure 4.4.13 with Figure 4.4.14 shows that the difference lies only in the table the object is attached to. The path, the slot number and the bytecode are the same.

This structure also shows that the cost of dynamic binding is small. A virtual call costs only one or two more memory reads than a plain call. Modern JVMs often remove even that. If only objects of the same class are seen at a call site, the component that compiles while the program runs (the JIT compiler) can bind the call directly to that body. So there is no need to avoid polymorphism for performance reasons.

4.4.4.5 What is not bound dynamically?

Dynamic binding applies only to instance methods that can be overridden. Fields, `static` methods, and `private` and `final` methods are bound statically: the compiler decides which to use by looking at the declared type of the reference.

```
class Animal {
    String kind = "mammal";

    static void info() {
        System.out.println("Animal.info()");
    }

    void introduce() {
        System.out.println("I am a " + kind);
    }
}

class Dog extends Animal {
    String kind = "dog"; // hides it

    static void info() {
        System.out.println("Dog.info()");
    }

    @Override
    void introduce() {
        System.out.println("I am a " + kind);
    }
}
```

These two classes show two different cases. `Dog` defines a new field called `kind`. This field does not override the `kind` field of `Animal`; it **hides** it: a `Dog` object has two separate `kind` fields. `Dog` also defines a `static info()` method with the same signature. `static` methods are not overridden either, they are hidden. `introduce()` is an ordinary instance method and is really overridden.

Definition 4.4.12

Defining in a subclass a field with the same name as a field in the superclass, or a `static` method with the same signature, is called **hiding**. The hidden member is not removed; it continues to exist in the object or the class. Which one is used is determined at compile time by the declared type of the reference.

```
Animal h = new Dog();
System.out.println(h.kind);
h.info();
h.introduce();
```

```
mammal
Animal.info()
I am a dog
```

Member	Chosen by	Binding
Instance method	Object type	Dynamic
Field (variable)	Reference type	Static
`static` method	Reference type	Static
`private` / `final` method	Cannot be overridden; there is one candidate	Static

Table 4.4.8: What is each kind of member chosen by?

A `final` method cannot be overridden in a subclass; if it is attempted, the compiler says `overridden method is final`. A `private` method is invisible to the subclass. If the subclass writes a method with the same name, it is a new, unrelated method. In both cases there is only one candidate for the call and the choice is known at compile time.

A `static` method belongs to the class, not to an object. Java allows a call through a reference such as `h.info()`, but this notation is misleading. The compiler quietly turns the call into `Animal.info()`; the object the reference points to is never looked at. Even if the reference is `null`, the call works and no `NullPointerException` occurs. That is why a `static` method is always called with the class name: `Dog.info()`. Defining a field with the same name as one in the superclass is also avoided.

4.4.4.6 Code trace: field hiding and `static`

The same idea can be traced step by step in a short program written with methods that return values.

```
class Animal {
    String kind = "mammal";
    static String info() { return "Animal.info"; }
    String introduce() { return "I am a " + kind; }
}
class Dog extends Animal {
    String kind = "dog";            // hides
    static String info() { return "Dog.info"; }
    String introduce() { return "I am a " + kind; }
}
Animal h = new Dog();   // main
System.out.println(h.kind);
System.out.println(h.info());
System.out.println(h.introduce());
System.out.println(((Dog) h).kind);
```

```
mammal
Animal.info
I am a dog
dog
```

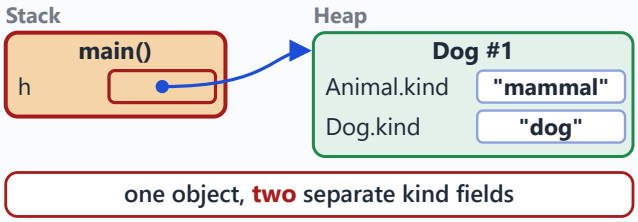


Figure 4.4.15: One object, two separate `kind` fields. The `kind` in `Dog` does not override the one in `Animal`; both are in the object.

Expression	Kind of member	Selection criterion	Selected	Output
`h.kind`	Field	Declared type `Animal`	`Animal.kind`	`animal`
`h.info()`	`static` method	Declared type `Animal`	`Animal.info()`	`Animal.info`
`h.introduce()`	Instance method	Actual type `Dog`	`Dog.introduce()`; inside it `kind` = `Dog.kind`	`I am a dog`
`((Dog)h).kind`	Field	Declared type `Dog` (by the cast)	`Dog.kind`	`dog`

Table 4.4.9: Trace of the field hiding program. The declared type of `h` is `Animal` and the object it points to is a `Dog`.

The third line needs extra attention. `h.introduce()` is bound dynamically and `Dog.introduce()` runs. The expression `kind` inside this method is also a field access, and it too is bound statically. But this time the declared type is the class in which the method is written. In code written in class `Dog`, `kind` means `this.kind`, and the type of `this` is `Dog`. So `Dog.kind` is read and `I am a dog` is printed.

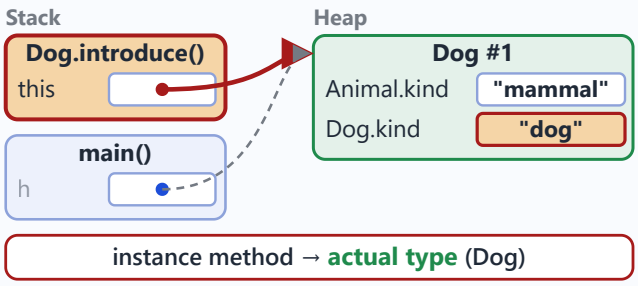


Figure 4.4.16: The call `h.introduce()`. The instance method is chosen by the actual type; the `this` inside `Dog.introduce()` has type `Dog` and reads `Dog.kind`.

For comparison, suppose `introduce()` were not overridden in `Dog`. Then `h.introduce()` would run the body in `Animal`. Since this body is written in class `Animal`, the `kind` inside it is `Animal.kind` and the output would be `I am a mammal`. The object is still a dog. This is the confusing side of field hiding: the same object holds two values under the same name, and which one is read depends on where the code is written.

On the last line the cast makes the declared type of the expression `Dog`. The other field of the same object is read: `dog`. The object has not changed; only the type the compiler looks at has.

> **Common mistake**
>
> Thinking fields are overridden too. `h.kind` is always read by the reference type. If information that varies by subclass is needed, the field is made `private` and a method (getter) that returns the information is written. A method can be overridden; a field cannot.

4.4.4.7 Calling an overridable method from a constructor

Dynamic binding works while an object is being built, too. This creates an unexpected trap: the constructor of a superclass can run a method that uses fields of the subclass that have not been prepared yet.

```
class Component {
    Component() {
        System.out.println("Building Component");
        draw();
    }
    void draw() { System.out.println("empty"); }
}
class Label extends Component {
    private String text = "Hello";
    Label() { System.out.println("Label built"); }
    @Override void draw() {
        System.out.println("Label: " + text);
    }
}
Label e = new Label();   // main
e.draw();
```

```
Building Component
Label: null
Label built
Label: Hello
```

The second line of the output is surprising. The field `text` was given the value `"Hello"` in its declaration, yet `Label: null` is printed. The reason is the order in which things are done while the object is built.

Step	Code running	`text`	Output
1	`new` reserves space for the object; the fields get their default values	`null`	—
2	The first job of `Label()` is the implicit `super()` call	`null`	—
3	Body of `Component()`, first line	`null`	`Building Component`
4	`draw()` = `this.draw()`; the object is a `Label` → `Label.draw()`	`null`	`Label: null`
5	`super()` returned; the subclass's field initializers run	`"Hello"`	—
6	Body of `Label()`	`"Hello"`	`Label built`
7	`main`: `e.draw()`	`"Hello"`	`Label: Hello`

Table 4.4.10: Trace of the call `new Label()`.

The critical point is the order of steps 4 and 5. In Java a subclass constructor first runs the superclass constructor. The initial values given to the subclass's fields in their declarations are assigned after the superclass constructor returns. The `draw()` call in `Component()` is bound dynamically; since the class of the object is `Label`, the overridden version is chosen. At that moment `text` is still at its default value `null`. The subclass's method runs on a half-built object.

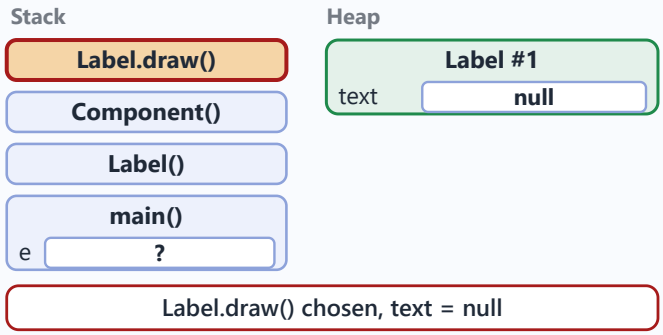


Figure 4.4.17: Step 4. On the stack are `Label()`, above it `Component()` and at the top `Label.draw()`. The `text` field of the object on the heap is still `null`.

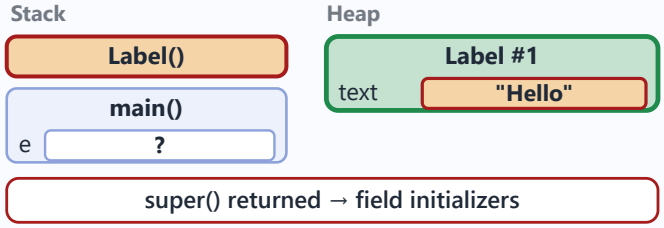


Figure 4.4.18: Step 5. After `super()` returns, the field initializer runs and `"Hello"` is assigned to `text`. `draw()` has long since been called.

In this example the harm is small because the method only prints the field. Had the method used the field, for example by calling `text.length()`, the program would have crashed:

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()"
because "this.text" is null
        at Label.draw(Main.java:21)
        at Component.<init>(Main.java:16)
        at Label.<init>(Main.java:19)
```

Since JDK 15 the `NullPointerException` message also says by default which expression was `null`: `"this.text" is null`. The order of the lines in the stack trace sums up the situation too. At the top is `Label.draw`, where the error occurred, below it the `Component` constructor that called it, and below that the `Label` constructor. `<init>` is the name of a constructor in bytecode. The chain starts at the constructor of `Label` and draws a loop that returns to a method of `Label`.

Definition 4.4.13

The constructor rule: Only `private`, `final` or `static` methods should be called from a constructor. These methods cannot be overridden; so they cannot reach the not-yet-built state of the subclass.

Common mistake

Calling an overridable method in a superclass constructor with a “template” intention (like `draw()`, `init()`, `load()`). If a subclass overrides this method, the method runs before the subclass's fields are assigned. The result is wrong output with `null`, `0` or `false` values, or a `NullPointerException`. If such preparation work is needed, it is called separately after the object is built.

4.4.5 Overloading

Method overloading is defining several methods with the same name but different parameter lists. It is a familiar tool from the topic of methods: `Math.abs` has `int`, `long`, `float` and `double` versions; `println` has ten versions. When used together with inheritance and dynamic binding, one property of overloading becomes important: the choice is made at compile time.

4.4.5.1 Overloading is resolved at compile time

```
static void print(Animal h) {
    System.out.println("print(Animal)");
}

static void print(Dog k) {
    System.out.println("print(Dog)");
}

public static void main(String[] args) {
    Dog dog = new Dog();
    Animal animal = dog;
    print(dog);
    print(animal);
    print((Dog) animal);
}
```

```
print(Dog)
print(Animal)
print(Dog)
```

All three calls of the program are made with the same object. Even so, the second call chooses a different version. In the first call the argument is `dog` and its declared type is `Dog`; both versions fit, but `print(Dog)` is more specific. In the second call the argument is `animal`. Even though the object is a dog, the declared type of `animal` is `Animal` and the compiler knows only that. An argument of type `Animal` does not fit `print(Dog)`; the only candidate is `print(Animal)`. In the third call the explicit cast makes the declared type of the expression `Dog`, and `print(Dog)` is chosen again.

Definition 4.4.14

The compiler chooses among overloaded versions by the **declared types** of the arguments. The chosen signature is written into the bytecode and does not change while the program runs. The actual class of the objects the arguments point to does not affect this choice.

This rule is the exact opposite of the rule for overriding. In overriding, the actual type of the object decides; in overloading, the declared type of the argument. Table 4.4.11 puts the two mechanisms side by side.

	Overloading	Overriding
Where?	Same class (or a class and its subclass)	Subclass
Signature	Different parameter list	Same signature
Who chooses	The compiler	The JVM

	Overloading	Overriding
Criterion	Declared type of the argument	Actual type of the receiver object
When	Compile time	Run time

Table 4.4.11: Comparison of overloading and overriding.

The phrase “receiver object” in the table needs a separate explanation. In the call `a.m(b)`, `a` is the receiver: it says on which object the method is called. `b` is the argument. In Java, dynamic binding looks only at the type of the receiver. There is no run-time choice that looks at the types of the arguments; for the arguments everything is settled at compile time. That is why Java's dynamic binding is called single dispatch.

One more detail concerns the return type. The signature of a method consists only of the name and the parameter types; the return type is not part of the signature. So two methods that differ only in return type cannot be overloaded. If `static int g(int x)` and `static long g(int x)` are written in the same class, the compiler says `method g(int) is already defined`.

4.4.5.2 The most specific version and ambiguity

An argument can fit several versions. In that case the compiler chooses the most specific one.

Definition 4.4.15

If every parameter type of one of two candidate methods is a subtype of (or the same as) the corresponding parameter type of the other, that method is **more specific**. The compiler chooses the most specific among the matching candidates. If there is no single most specific candidate, the call is **ambiguous** and gives a compile error.

```
static void show(Object o) { System.out.println("show(Object)"); }
static void show(Animal h) { System.out.println("show(Animal)"); }
static void show(String s) { System.out.println("show(String)"); }
// in main:
show(new Cat());
show("a");
Object o = "a";
show(o);
show((Animal) null);
```

```
show(Animal)
show(String)
show(Object)
show(Animal)
```

Let us examine the four calls in order. `show(new Cat())`: a cat fits both `Animal` and `Object`, not `String`. `Animal` is a subtype of `Object`, so it is more specific: `show(Animal)`. `show("a")`: the argument fits the `String` and `Object` versions; `String` is more specific. `show(o)`: even though the object `o` points to is a string, its declared type is `Object`. An argument of type `Object` fits only `show(Object)`.

The fourth call uses a special value. `null` fits every reference type; so when `show(null)` is written, all three versions are candidates. `Object` is the most general and is eliminated. But there is no inheritance relationship between `Animal` and `String`; neither is more specific than the other. The compiler cannot choose:

```
error: reference to show is ambiguous
  both method show(Animal) in Main and
  method show(String) in Main match
```

The solution is to fix the declared type of the expression with an explicit cast: `show((Animal) null)`. The value of the expression `(Animal) null` is still `null`, but its type is `Animal`. Now the `String` version does not fit and `show(Animal)` is chosen. This is the fourth line in the output.

If the matching versions lie on a single inheritance chain, there is no ambiguity. With `show(Object)`, `show(Animal)` and `show(Cat)`, `show(null)` compiles and chooses the lowest one, `show(Cat)`, because `Cat` is a subtype of both other types. Ambiguity arises only when two candidates that are not subtypes of each other fit at the same time.

Common mistake

Passing `null` when there are two unrelated overloads: `show(null)` → `reference to show is ambiguous`. Which version is wanted is stated with an explicit cast: `show((String) null)` or `show((Animal) null)`.

4.4.5.3 Code trace: the compiler or the JVM?

The most instructive case is the one where the two mechanisms work together in the same call. In the program below `print` is overloaded; in each version the overridden `makeSound()` is also called.

```
static void print(Animal h) {
    System.out.print("print(Animal): ");
    h.makeSound();
}
static void print(Dog k) {
    System.out.print("print(Dog): ");
    k.makeSound();
}
// in main:
Animal animal = new Dog();
Animal other = new Cat();
print(animal);
print(other);
print((Dog) animal);
```

```
print(Animal): Woof woof!
print(Animal): Meow!
print(Dog): Woof woof!
```

Call	Declared type of the argument	Compiler's choice	Object	JVM's choice	Output
`print(animal)`	`Animal`	`print(Animal)`	`Dog`	`Dog.makeSound()`	`print(Animal): Woof woof!`
`print(other)`	`Animal`	`print(Animal)`	`Cat`	`Cat.makeSound()`	`print(Animal): Meow!`
`print((Dog) animal)`	`Dog`	`print(Dog)`	`Dog`	`Dog.makeSound()`	`print(Dog): Woof woof!`

Table 4.4.12: Two decisions in every call. The compiler chooses the version of `print`, the JVM the version of `makeSound()`.

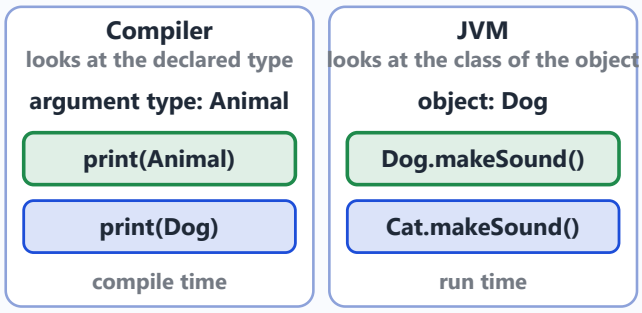


Figure 4.4.19: Two decisions in the first call. The compiler looked at the declared type (`Animal`) and chose `print(Animal)`; inside the method the JVM looks at the object (`Dog`) and chooses `Dog.makeSound()`.

In the first two calls the declared types of the variables are the same, and the objects they point to differ. The compiler chooses the same version, `print(Animal)`, in both calls. The `h.makeSound()` inside the method is an overridden method, and the JVM looks at the object each time: first `Woof woof!`, then `Meow!`. In the third call the explicit cast changes the declared type of the expression and the compiler chooses `print(Dog)`.

The trace can be summed up in one sentence: the overloading choice is made at compile time with the declared type, the overriding choice at run time with the actual type. To find out what a call prints, two questions are asked in order. First, "which signature does the compiler choose?", then "which class's body of that signature runs?".

4.4.5.4 The three phases of overload resolution

When the argument is a primitive value, the choice gets one step more complicated. An `int` can be widened to `long`; it can be boxed to `Integer`; it can also be passed to an `int...` parameter. The compiler tries these routes not at the same time but in order.

Definition 4.4.16

The compiler searches overloaded versions in **three phases**:

1. **The phase without boxing.** Only an exact match, primitive widening (`int` → `long`) and reference widening (`String` → `Object`) are tried.
2. **The boxing phase.** If there is no candidate in phase 1, automatic boxing and unboxing are tried too: `double` → `Double` → `Object`.
3. **The variable-arity phase.** If there is still no candidate, versions with variable arity such as `int...` come into play.

If at least one candidate is found in a phase, later phases are not reached. The most specific among that phase's candidates is chosen.

The most surprising consequence of the rule is this: with `f(long)` and `f(Integer)`, the call `f(5)` chooses `f(long)`. `Integer` is the wrapper class of `int` and looks closer at first sight. But reaching it needs boxing; `long` is reached by widening, in phase 1. Widening comes before boxing for historical reasons. Boxing was added in Java 5. Until then `f(5)` called `f(long)`. Boxing was put in the second phase so that the new rule would not change the behavior of old programs.

Narrowing (`long` → `int`, `double` → `long`) is tried in no phase. If no candidate is found in any of the three phases, the compiler says `no suitable method found`. For example, with only `print(Animal)` and `print(Dog)`, writing `print(true)` gives the message `no suitable method found for print(boolean)`.

This order is applied at compile time and the chosen signature is written into the bytecode. Which class's body of the chosen signature runs is again determined at run time. The three phases answer only the question “which signature?”.

```
static void f(long x)    { p("f(long)"); }
static void f(Integer x) { p("f(Integer)"); }
static void f(Object x)  { p("f(Object)"); }
static void f(int... x)  { p("f(int...)"); }
// in main (p: System.out.println):
byte b = 1;
f(5);
f(b);
f(Integer.valueOf(5));
f("x");
f(2.5);
f();
f(1, 2);
```

```
f(long)
f(long)
f(Integer)
f(Object)
f(Object)
f(int...)
f(int...)
```

Call	Argument type	Phase 1 candidates	Phase 2	Phase 3	Chosen
`f(5)`	`int`	`f(long)` (widening)	—	—	`f(long)`
`f(b)`	`byte`	`f(long)` (widening)	—	—	`f(long)`
`f(Integer.valueOf(5))`	`Integer`	`f(Integer)`, `f(Object)`	—	—	`f(Integer)` (more specific)
`f("x")`	`String`	`f(Object)`	—	—	`f(Object)`
`f(2.5)`	`double`	none	`f(Object)` (boxing to `Double`)	—	`f(Object)`
`f()`	—	none	none	`f(int...)`	`f(int...)`
`f(1, 2)`	`int`, `int`	none	none	`f(int...)`	`f(int...)`

Table 4.4.13: Resolving seven calls phase by phase.

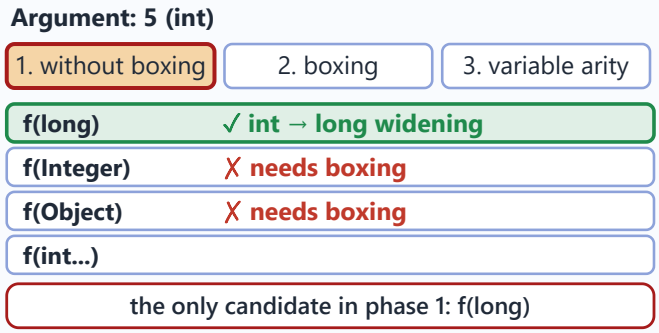


Figure 4.4.20: The phases for `f(5)`. In phase 1 only `long` fits, by widening; `Integer` and `Object` need boxing and are not tried in this phase.

A few rows of the table need a separate explanation. In `f(b)` the argument is a `byte`, and it also widens to `long`. Even if it were boxed, it would become a `Byte`, not an `Integer`; there is no relationship between `Byte` and `Integer`. In `f(Integer.valueOf(5))` the argument is already an object. In phase 1 there are two candidates: `f(Integer)` by exact match and `f(Object)` by reference widening. `Integer` is chosen because it is more specific than `Object`. `f(long)` needs unboxing and is not tried in this phase.

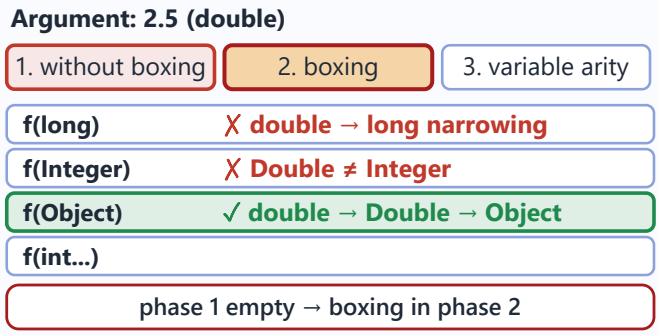


Figure 4.4.21: The phases for `f(2.5)`. A `double` cannot be narrowed to `long`; phase 1 stays empty. In phase 2 it is boxed to `Double` and fits `Object`.

The line `f(2.5)` shows that narrowing is never tried. Going from `double` to `long` loses information; the compiler does not do it by itself. When phase 1 stays empty, phase 2 is entered. `2.5` is boxed to `Double`. `Double` does not fit `Integer`, but it fits `Object`. In the last two calls there are zero or two arguments. Only the `int...` parameter accepts these counts, and it is consulted last, in phase 3.

Example 4.4.4

For the same four versions (`f(long)`, `f(Integer)`, `f(Object)`, `f(int...)`), which versions do `short s = 3; f(s);`, `f('A');` and `f(Long.valueOf(4));` choose?

`f(s)`: `short` widens to `long`; in phase 1 the only candidate is `f(long)`.

`f('A')`: `char` is also an integral type and widens to `long`: `f(long)`. The code of the character, 65, is passed as a `long`.

`f(Long.valueOf(4))`: the argument is a `Long` object. `f(long)` needs unboxing and is not tried in phase 1. `Long` does not fit `Integer`. In phase 1 the only candidate, by reference widening, is `f(Object)`.

Common mistake

Thinking boxing comes first. With `f(long)` and `f(Integer)`, `f(5)` calls `f(long)`, not `f(Integer)`: widening is in phase 1, boxing in phase 2.

4.4.6 Inheritance and polymorphism

The real value of polymorphism shows in the design of a program. Polymorphic code is shorter than code that tests types and more robust against new classes. Most of the examples are built on a shape hierarchy.

4.4.6.1 The relationship between inheritance and polymorphism

The relationship between the two concepts is one-directional. Run-time polymorphism requires inheritance: without a class hierarchy there is neither overriding nor dynamic binding. Using inheritance, on the other hand, does not require polymorphism. A class can be derived from another just to reuse code. Inheritance builds the hierarchy; polymorphism makes use of that hierarchy.

If class `B` is a subclass of `A`, an object of `B` can behave like an object of `A`; the reverse is not true. Assigning an `A` reference to a `B` reference needs a narrowing cast, and the cast is written in parentheses.

```
A objA = new B(); // B acts as an A
B objB = (B) objA; // legal, but
                   // use with care
```

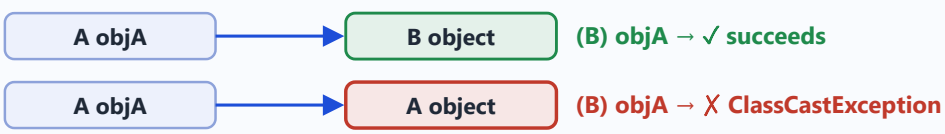


Figure 4.4.22: The two possible results of the cast `(B) objA`. If `objA` points to a `B` object, the cast succeeds; if it points to a plain `A` object, `ClassCastException` is thrown.

The last line compiles. At run time, if `objA` does not really point to a `B` object, `ClassCastException` is thrown. The “use with care” comment in the code describes this risk.

In Java, class inheritance is not the only source of polymorphism. Interfaces provide the same mechanism. The object of any class that implements an interface (`implements`) can be assigned to a reference of that interface type. A class can derive from only one superclass, but it can implement many interfaces. The same dynamic binding works with an interface type such as `SoundMaker s` instead of `Animal h` .

4.4.6.2 A shape hierarchy

Suppose a drawing program has squares, rectangles and circles. Every shape has a name; every shape's area and perimeter can be computed; every shape can be drawn. But area and perimeter are found with a different formula in each shape. This common structure is gathered in an abstract superclass.

```
abstract class Shape {
    private String name;

    public Shape(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public abstract double area();

    public abstract double perimeter();

    public void draw() {
        System.out.println(name + " drawn.");
    }
}
```

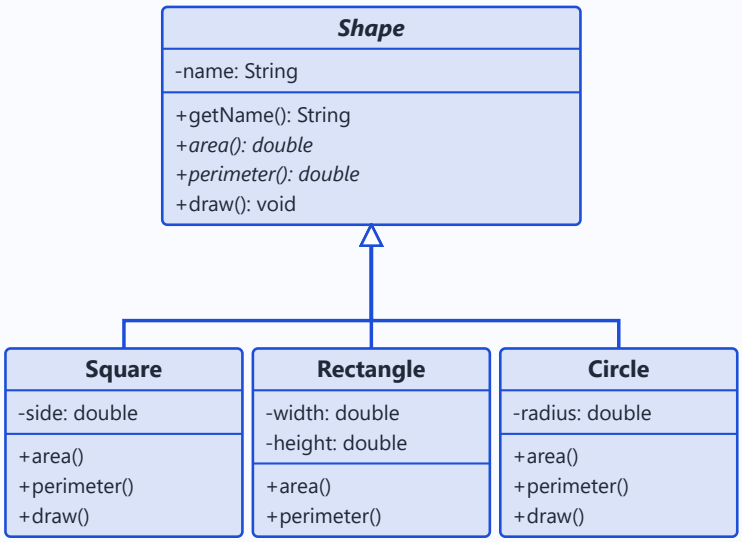


Figure 4.4.23: The abstract class `Shape` and its subclasses `Square`, `Rectangle`, `Circle`. Names and methods in italics are abstract. `Rectangle` does not override `draw()`.

The word `abstract` at the head of class `Shape` declares that the class is abstract. An object cannot be created directly from an abstract class; if `new Shape("x")` is written, the compiler says `Shape is abstract; cannot be instantiated`. `area()` and `perimeter()` are abstract methods too: they have no body and every concrete subclass must write them. `draw()` is an ordinary method with a body. Subclasses can use it as it is or override it. The `name` field is `private`; subclasses reach it through the constructor and `getName()`.

An abstract class is an ideal superclass for polymorphism. The question “what is the area of a shape?” has no general answer; only a concrete shape can give it. An abstract method expresses this situation: “every shape has an area, but the subclass says how it is computed.”

```
class Square extends Shape {
    private double side;
    public Square(double side) {
        super("Square");
        this.side = side;
    }
    @Override
    public double area() {
        return side * side;
    }
    @Override
    public double perimeter() {
        return 4 * side;
    }
    @Override
    public void draw() {
        System.out.println("Square drawn: side " + side);
    }
}
```

The constructor of `Square` passes the name to the superclass with `super("Square")` and keeps the side length in its own field. It overrides all three methods. If `Square` did not write the abstract methods, it would not compile unless it were declared `abstract`.

```
class Rectangle extends Shape {
    private double width;
    private double height;
    public Rectangle(double width, double height) {
        super("Rectangle");
        this.width = width;
        this.height = height;
    }
    @Override
    public double area() {
        return width * height;
    }
    @Override
    public double perimeter() {
        return 2 * (width + height);
    }
    // no draw() → Shape's is used
}
```

`Rectangle` overrides area and perimeter but does not override `draw()`. When a rectangle is drawn, `Shape.draw()` runs and prints `Rectangle drawn.` using the name. The comment at the end of the code reminds of this.

```
class Circle extends Shape {
    private double radius;
    public Circle(double radius) {
        super("Circle");
        this.radius = radius;
    }
    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
    @Override
    public double perimeter() {
        return 2 * Math.PI * radius;
    }
    @Override
    public void draw() {
        System.out.println("Circle drawn: radius " + radius);
    }
}
```

`Circle` computes the area and perimeter with the constant `Math.PI` and overrides `draw()` with its own information, printing the radius too.

4.4.6.3 Code trace: an array of shapes

```
Shape[] shapes = new Shape[3];
shapes[0] = new Square(5);
shapes[1] = new Rectangle(4, 6);
shapes[2] = new Circle(7);

for (Shape shape : shapes) {
    System.out.println(shape.getName()
        + " area: " + shape.area());
    shape.draw();
}
```

```
Square area: 25.0
Square drawn: side 5.0
Rectangle area: 24.0
Rectangle drawn.
Circle area: 153.93804002589985
Circle drawn: radius 7.0
```

The array has type `Shape`; its elements point to objects of three different subclasses. In every round the loop makes the same two calls: `shape.area()` and `shape.draw()`. `getName()` is not overridden in any subclass; in every round the version of `Shape` runs and returns the name given in the constructor.

Round	Object	`area()`	Area	`draw()`
1	`Square(5)`	`Square`: 5 × 5	25.0	`Square.draw()`
2	`Rectangle(4, 6)`	`Rectangle`: 4 × 6	24.0	`Shape.draw()` (not overridden)
3	`Circle(7)`	`Circle`: π × 7 × 7	153.93804002589985	`Circle.draw()`

Table 4.4.14: Trace of the shape array loop.

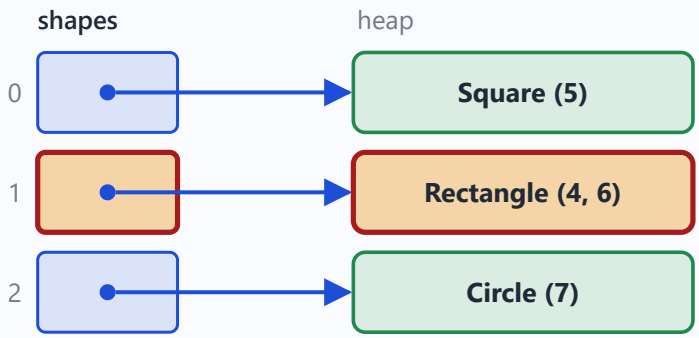


Figure 4.4.24: The second round. `shapes[1]` points to a `Rectangle`. Since `Rectangle` does not override `draw()`, the search climbs up and `Shape.draw()` runs.

The area of the circle is 49π. `double` arithmetic computes this value as 153.93804002589985, and `println` prints all significant digits. If a more readable output is wanted, `printf("%.2f", ...)` is used.

4.4.6.4 Why polymorphism?

The value of polymorphism is seen by comparison with code written without it. The method below finds the area of a shape by testing types.

```
static double badArea(Shape s) {
    double area;
    if (s instanceof Square) {
        area = ((Square) s).area();
    } else if (s instanceof Rectangle) {
        area = ((Rectangle) s).area();
    } else if (s instanceof Circle) {
        area = ((Circle) s).area();
    } else {
        area = 0; // new type forgotten!
    }
    return area;
}
```

The method works correctly, but it is fragile. When a new shape is added to the hierarchy, this method must change too. If a class `RightTriangle extends Shape` is written and the method is not updated, `badArea(new RightTriangle(3, 4))` returns `0.0` without any error. Every `instanceof` chain in a program is a place that can be forgotten when a new class is added. Moreover, the casts in the method are unnecessary: `area()` is already defined in `Shape` and can be called without a cast.

```
static double totalArea(Shape[] arr) {
    double total = 0;
    for (Shape shape : arr) {
        // each object knows its area
        total += shape.area();
    }
    return total;
}
```

In the polymorphic version there is no type test at all. Each object computes its own area. It is enough to add a new class `RightTriangle extends Shape`; `totalArea` does not change at all. For example, for the array `{Square(5), Rectangle(4, 6), RightTriangle(3, 4)}` the result is 25 + 24 + 6 = **55.0**. The code gets shorter and the risk of `ClassCastException` disappears.

Definition 4.4.17

The open/closed principle: A software unit should be open for extension and closed for modification. New behavior is added not by changing existing code but by adding new code (for example a new subclass).

`totalArea` follows this principle: it extends with new shapes but does not change itself. `badArea` breaks the principle. In large programs the difference is much more marked. A shape editor may have dozens of operations such as area, perimeter, drawing, saving and printing. In a design that tests

types, a new shape requires touching all of these operations; in a polymorphic design only the new class is written.

Example 4.4.5

Let `RightTriangle(a, b)` be added to the shape hierarchy: area = a·b/2, perimeter = a + b + hypotenuse. Let `draw()` not be overridden and the constructor call `super("Right triangle")`. For `RightTriangle(3, 4)`, what are the area, the perimeter and the output of `draw()`?

Area: $3 \cdot 4 / 2 = \mathbf{6.0}$. Hypotenuse: $\sqrt{(3^2 + 4^2)} = \sqrt{25} = 5$, perimeter: $3 + 4 + 5 = \mathbf{12.0}$. Since `draw()` is not overridden, `Shape.draw()` runs: `Right triangle drawn.`

Nothing changes in the shape loop; only a new object is added to the array. For the perimeter, `a + b + Math.sqrt(a * a + b * b)` is written in the code.

4.4.6.5 Polymorphic parameters and return values

Polymorphism is used not only in arrays but also in method parameters and return values.

```
class Vet {
    void examine(Animal h) {     // any subclass
        System.out.print("Exam: ");
        h.makeSound();
    }
    static Animal adopt(String kind) { // hidden
        if (kind.equals("cat")) return new Cat();
        return new Dog();
    }
}
// in main:
Vet v = new Vet();
v.examine(new Cat());
Animal h = Vet.adopt("dog");
v.examine(h);
System.out.println(h.getClass().getSimpleName());
```

```
Exam: Meow!
Exam: Woof woof!
Dog
```

The parameter of the `examine` method has type `Animal`. So the method accepts objects of every subclass that exists today and that will be written in the future. In the first call a cat is examined, in the second a dog; the `h.makeSound()` inside the method runs the object's own version every time.

When a new class `Bird` is added to the hierarchy, `examine` does not change.

The return type of the `adopt` method is `Animal` too. The method hides from the caller which subclass it produces. The caller relies only on the `Animal` contract: the object returned can make a sound, run and sleep. If `adopt` starts returning another subclass later, the code that uses it is not affected. The `getSimpleName()` on the last line gives the name of the actual class of the object without the package name: `Dog`.

Definition 4.4.18

A **factory method** is a method that produces an object and returns it as a superclass or interface type. The caller does not have to know which concrete class was produced.

A general principle comes out of this example: parameter and variable types are chosen as the most general type that is enough to do the job. Since `examine` only calls `makeSound()`, a parameter of type `Animal` is enough; a parameter of type `Dog` would leave the cat out needlessly. The same principle is seen in the Java library: variables are mostly declared with type `List<String>` rather than `ArrayList<String>`.

4.4.6.6 The anti-tank game

The use of polymorphism in a game is a more realistic example. In an anti-tank game written with Swing, tanks fall from the top of the screen; the player fires shells at them with the anti-tank gun at the bottom. When the score is a multiple of 10, the type of the gun changes and it starts firing a different shell.

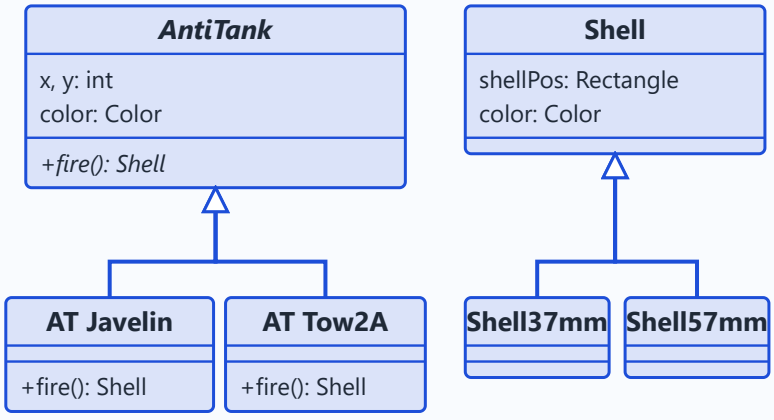


Figure 4.4.25: The two hierarchies of the anti-tank game. Two subclasses override the `fire()` method of the abstract class `AntiTank`; class `Shell` also has subclasses `Shell37mm` and `Shell57mm`.

In the figure, “AT Javelin” is the class `AntiTankJavelin` and “AT Tow2A” is the class `AntiTankTow2A`. Class `Tank` also has subclasses `TankT90MS` and `TankLeopard2A4`. The game loop knows only the types `AntiTank` and `Tank`.

```
AntiTank weapon = new AntiTankJavelin(x, y);
...
// which shell? depends on the object (dynamic binding)
shells.add(weapon.fire());
...
if (weapon instanceof AntiTankJavelin) {
    weapon = new AntiTankTow2A(weapon.x, weapon.y);
}
```

The line `shells.add(weapon.fire())` does not know which shell will be produced. `fire()` is an abstract method; the call is bound dynamically and the version of whatever class the gun is at that moment runs. The return type is `Shell`; the object returned may be `Shell37mm` or `Shell57mm`. The `shells` list also holds objects of type `Shell` and moves and draws all of them with the same loop.

The `instanceof` test in the code does a different job: upgrading the gun. The test is made only once, at upgrade time; not at every shot. The new gun object takes the position (`x`, `y`) of the old one and the variable `weapon` now points to it. The declared type of the variable has not changed; only the object it points to has. From this moment on the same `fire()` call runs another body.

```
Shell t1 = weapon.fire(); // Javelin
// → Shell37mm, height 30
weapon = new AntiTankTow2A(100, 540);
Shell t2 = weapon.fire();
// → Shell57mm, height 40
```

A test output independent of the game confirms this. The same `fire()` call produces a different shell when the gun changes: Javelin fires a `Shell37mm` with height 30, Tow2A a `Shell57mm` with height 40. No `if` on the type of the shell is needed in the game code. To add a new kind of gun, only a new subclass and a new shell class are written.

4.4.6.7 `equals` and inheritance: the symmetry trap

The methods of `Object` are bound dynamically too. When `Object o = k1; System.out.println(o);` is written, `println` calls `o.toString()` and the object's own `toString` runs. `o.equals(x)` likewise runs the object's own `equals`. This makes collections work correctly with every class. Methods such as `ArrayList.contains` and `HashSet.add` know only the type `Object`, but they call the elements' own `equals` and `hashCode` methods.

The `equals` method must obey certain rules. One of them is **symmetry**: if `x.equals(y)` is true, `y.equals(x)` must be true too. Once inheritance is involved, this rule is easily broken.

```
class Point {
    final int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
    @Override public boolean equals(Object o) {
        if (!(o instanceof Point)) return false;
        Point n = (Point) o;
        return x == n.x && y == n.y;
    }
    @Override public int hashCode() { return 31 * x + y; }
}
class ColoredPoint extends Point {
    final String color;
    ColoredPoint(int x, int y, String color) {
        super(x, y);
        this.color = color;
    }
    @Override public boolean equals(Object o) {
        if (!(o instanceof ColoredPoint)) return false;
        ColoredPoint r = (ColoredPoint) o;
        return super.equals(r) && color.equals(r.color);
    }
}
```

`Point` counts two points equal if their coordinates are equal. `ColoredPoint` adds the color condition to this and requires the other side to be a `ColoredPoint` as well. Each of the two classes is reasonable when looked at separately. A problem arises when they are used together:

```
Point p = new Point(1, 2);
ColoredPoint k =
    new ColoredPoint(1, 2, "red");
System.out.println(p.equals(k)
    + " " + k.equals(p));
```

```
true false
```

The receiver object determines which `equals` runs. In the call `p.equals(k)` the receiver is `p` and the method of `Point` runs. `k` is a `ColoredPoint`, therefore a `Point`; the coordinates are equal and the result is `true`. In the call `k.equals(p)` the receiver is `k` and the method of `ColoredPoint` runs. `p` is not a `ColoredPoint`; the result is `false`. Symmetry is broken.

This breakage causes inconsistent behavior in collections. In a list that contains `k`, `contains(p)` gives `true`, because the list calls `p.equals(k)`. In a list that contains `p`, `contains(k)` gives `false`. If `p` and then `k` are added to a `HashSet`, the set has 2 elements. If `k` and then `p` are added, it has 1 element. The contents of the set have come to depend on the order in which the elements were added.

Operation	Called inside	Result
`[k].contains(p)`	`p.equals(k)`	`true`
`[p].contains(k)`	`k.equals(p)`	`false`
`HashSet` : first `p` , then `k`	`k.equals(p)`	2 elements
`HashSet` : first `k` , then `p`	`p.equals(k)`	1 element

Table 4.4.15: Consequences of broken symmetry in collections. `p` = `Point(1, 2)` , `k` = `ColoredPoint(1, 2, "red")` .

There are two known solutions to the problem. The first is to use a `getClass()` comparison instead of `instanceof` in both classes. That way only objects of exactly the same class count as equal, and `p.equals(k)` is `false` too:

```
// in Point:
@Override public boolean equals(Object o) {
    if (o == null || getClass() != o.getClass()) return false;
    Point n = (Point) o;
    return x == n.x && y == n.y;
}
// in ColoredPoint:
@Override public boolean equals(Object o) {
    if (!super.equals(o)) return false;   // class and coordinates
    return color.equals(((ColoredPoint) o).color);
}
```

With this solution `p.equals(k)` and `k.equals(p)` both give `false` ; two points of the same color come out equal. The price is this: even an ordinary subclass of `Point` that adds no field is no longer counted equal to a `Point` .

The second solution is composition instead of inheritance. `ColoredPoint` does not derive from `Point` ; it holds a `Point` field and carries a color as well. Since there is no is-a relationship between the two classes, comparing them with each other also loses its meaning. If the position of a colored point is needed, it is obtained as a `Point` with a method such as `position()` .

```
class ColoredPoint {              // no extends
    private final Point position;
    private final String color;
    ColoredPoint(int x, int y, String color) {
        this.position = new Point(x, y);
        this.color = color;
    }
    Point position() { return position; }
    @Override public boolean equals(Object o) {
        if (!(o instanceof ColoredPoint)) return false;
        ColoredPoint r = (ColoredPoint) o;
        return position.equals(r.position) && color.equals(r.color);
    }
    @Override public int hashCode() { return 31 * position.hashCode() + color.hashCode(); }
}
```

Common mistake

Extending `equals` in a subclass in one direction only. If the subclass adds a new condition to `equals` while the superclass keeps the old rule, `p.equals(k)` is `true` and `k.equals(p)` is `false`. Symmetry breaks and collections become inconsistent.

4.4.7 Worked problems

The problems below require using the rules explained in the notes together. In each problem the code is given first, then the solution is explained step by step. All outputs were verified by running them with JDK 21.

Example 4.4.6

Problem 1: What is the output? `call()` is defined only in `A` and calls `print()` from inside. `B` overrides only `print()`.

```
class A {
    void print() {
        System.out.println("A.print");
    }

    void call() {
        print(); // this.print()
    }
}

class B extends A {
    @Override
    void print() {
        System.out.println("B.print");
    }
}

// in main:
A a = new B();
a.call();
new A().call();
```

`a.call()`: the object `a` points to is a `B`. `B` has no `call()`; the search climbs up and `A.call()` runs. The `print()` inside this method is really `this.print()`. `this` is the object the method was called on, that is a `B` object. `this.print()` is bound dynamically too and `B.print()` runs: `B.print`.

`new A().call()`: the object is an `A`; `this.print()` inside `A.call()` runs `A.print()`: `A.print`.

```
B.print
A.print
```

The lesson of the problem is this: the code of a superclass may unknowingly call a method that a subclass overrides. The programmer who wrote `A` may have thought that `call()` would call `A.print()`. But every call made through `this` is bound dynamically.

Example 4.4.7

Problem 2: Does it compile, does it run? For each line, the verdict is to be given: compile error, run-time error, or no problem (with its output).

```
Animal h = new Cat();
Cat k1 = h;              // (1)
Cat k2 = (Cat) h;        // (2)
Dog k3 = (Dog) h;        // (3)
h.scratch();             // (4)
((Cat) h).scratch();     // (5)
Object o = new Dog();
o.makeSound();           // (6)
((Animal) o).makeSound(); // (7)
```

#	Result	Reason
1	Compile error	`Animal` → `Cat` needs an explicit cast: `incompatible types: Animal cannot be converted to Cat`.
2	No problem	The object really is a `Cat`.
3	Run-time error	It compiles, but the object is a cat: `ClassCastException`.
4	Compile error	`Animal` has no `scratch()`: `cannot find symbol`.
5	No problem	Prints `Cat scratches`.
6	Compile error	`Object` has no `makeSound()`.
7	No problem	Dynamic binding: prints `Woof woof!`.

The method of solution is the same for every line. First the compiler's question is asked: is the assignment or call valid by the declared types? If it is, the JVM's question is asked: looking at the object, does the cast succeed, and which version runs? In lines 1 and 4 the compiler knows that `h` is an `Animal`; it does not know it is a cat. In line 7 the cast is downward (Object → Animal) and the object is a dog; `Dog.makeSound()` runs.

Example 4.4.8

Problem 3: Overloading and overriding. `Parent` has two overloaded `f` methods; `Child` overrides only `f(Object)`. What do the two calls print?

```
class Parent {
    void f(Object o) {
        System.out.println("Parent.f(Object)");
    }

    void f(String s) {
        System.out.println("Parent.f(String)");
    }
}

class Child extends Parent {
    @Override
    void f(Object o) {
        System.out.println("Child.f(Object)");
    }
}

// in main:
Parent u = new Child();
Object o = "hello";
u.f(o);
u.f("hello");
```

`u.f(o)`: first the compiler chooses the version. The declared type of `u` is `Parent`; the candidates are `f(Object)` and `f(String)`. The declared type of `o` is `Object`; only `f(Object)` fits. That the object it points to is a string does not change this choice. At run time the receiver object is a `Child` and has overridden `f(Object)`: `Child.f(Object)`.

`u.f("hello")`: the argument is a `String`; both versions fit, and `f(String)` is more specific. Since `Child` does not override this version, `Parent.f(String)` runs.

```
Child.f(Object)
Parent.f(String)
```

The order is always the same: first overloading (compile time, declared type), then overriding (run time, object type).

Example 4.4.9

Problem 4: Construction, phase, array. For the classes below, what do `new Child();` and then `new Child().show();` print?

```
class Parent {
    Parent() { show(); }
    void show() { System.out.println("Parent"); }
}
class Child extends Parent {
    private int[] arr = {1, 2, 3};
    @Override void show() {
        System.out.println(arr == null
            ? "no arr" : "length " + arr.length);
    }
}
```

During every `new Child()`, `Parent()` runs first and calls `show()`. The call is bound dynamically; `Child.show()` runs. At that moment the field `arr` has not yet been given its initial value and is `null`: `no arr`. On the second line a new object is created and the same thing happens again: `no arr` again. Then `show()` is called on the built object; now `arr` is ready: `length 3`. The output is three lines: `no arr`, `no arr`, `length 3`.

The second question of the same problem: do `Object[] d = new String[1]; d[0] = 1;` compile, do they run? They compile: a `String[]` can be assigned to an `Object[]` reference, and `1` can be boxed to `Integer` and written as an `Object`. At run time, however, the array is really a `String[]`: `ArrayStoreException: java.lang.Integer`.

Example 4.4.10

Problem 5: Salaries. The `salary()` method of class `Employee` returns the base salary. `Manager` overrides it with base × 1.5, `Intern` with a fixed 5000. For `Employee[] = {Ayşe 20000, Manager Mehmet 30000, Intern Can}` what is the total salary?

```
class Manager extends Employee {
    Manager(String name, double baseSalary) { super(name, baseSalary); }
    @Override double salary() { return baseSalary * 1.5; }
}
class Intern extends Employee {
    Intern(String name) { super(name, 0); }
    @Override double salary() { return 5000; }
}
// in main:
Employee[] cs = { new Employee("Ayşe", 20000), new Manager("Mehmet", 30000),
                  new Intern("Can") };
double total = 0;
for (Employee c : cs) total += c.salary();
System.out.println(total);   // 70000.0
```

The loop knows only the type `Employee`; in every round `c.salary()` runs the object's own version. Ayşe is a plain employee: 20000. Mehmet is a manager: 30000 × 1.5 = 45000. Can is an intern: 5000. The total is 20000 + 45000 + 5000 = **70000.0**. Since the result is a `double`, it is printed with a decimal point.

Example 4.4.11

Problem 6: X, Y, Z. In `X`, `name()` returns `"X"` and `print()` does `println(name())`. `Y extends X` overrides only `name()` to return `"Y"`. `Z extends Y` overrides only `print()` as `println("Z:" + name())`. If `print()` is called for every element of the array `{new X(), new Y(), new Z()}`, what is the output?

`X: X.print()` → `X.name()`: `X`. `Y: Y` has no `print()`, so `X.print()` runs; but the `name()` inside it, because `this` is a `Y`, is `Y.name()`: `Y`. `Z: Z.print()` runs; `Z` has no `name()`, the search ends at `Y: Z:Y`.

```
X
Y
Z:Y
```

Example 4.4.12

Problem 7: Removing the `instanceof` chain. How can the loop with the `instanceof` chain (`Dog` → bark, `Cat` → scratch, other → `"No special behavior"`) be written without any `instanceof`?

A method `void special()` is added to `Animal` and prints `"No special behavior"`. `Dog` overrides it to call `bark()`, and `Cat` to call `scratch()`. The loop shrinks to a single line:

```
// in Animal:
void special() { System.out.println("No special behavior"); }
// in Dog:
@Override void special() { bark(); }
// in Cat:
@Override void special() { scratch(); }
// loop:
for (Animal x : animals) {
    x.special();
}
```

The output is the same three lines. When a new subclass is added, the loop does not change; the new class overrides `special()` if it wants to, and otherwise gets the default behavior.

Example 4.4.13

Problem 8: A call through an `Object` reference. Why does `Object o = new Dog();` followed by `o.makeSound();` not compile? With `print(Animal)` and `print(Dog)` present, does `print(o)` compile?

`Object` has no `makeSound()`: `cannot find symbol … location: variable o of type Object`. The fix: `((Animal) o).makeSound();`. Since the object is a dog, it prints `Woof woof!`.

`print(o)` does not compile either. The declared type of `o` is `Object`, and neither version accepts an `Object`: `no suitable method found for print(Object)`. With `print((Dog) o)` it compiles and `print(Dog)` is chosen.

4.4.8 Common mistakes

Mistakes related to polymorphism fall into two groups. Compile errors come from the compiler looking only at the declared type, and their messages are clear. Run-time and logic errors pass the compiler; they show up either as an exception or as silently wrong results. The two tables below gather the mistakes treated separately in the body of these notes. The messages were obtained with the javac and java tools of JDK 21.

Code	javac message	Reason / fix
`Dog k = new Animal();`	`incompatible types: Animal cannot be converted to Dog`	Not every animal is a dog; the reverse assignment is not possible.
`Dog k = animal;`	`incompatible types: Animal cannot be converted to Dog`	Even if the object is a dog, an explicit cast is needed: `(Dog) animal`.
`animal.bark();`	`cannot find symbol … variable animal of type Animal`	The method is not in the reference type; a downcast is needed first.
`(Dog) h.bark();`	`not a statement`	Missing parentheses: `((Dog) h).bark();`
`(Cat) dog`	`incompatible types: Dog cannot be converted to Cat`	There is no cast between sibling classes.
`@Override void makeSound()`	`method does not override or implement a method from a supertype`	The name is misspelled (upper/lower case).
`void makeSound()` (`public` in the superclass)	`attempting to assign weaker access privileges; was public`	An overriding method cannot narrow access.
`show(null)` (two unrelated versions)	`reference to show t ambiguous`	Chosen with an explicit cast: `show((Animal) null)`.

Code	javac message	Reason / fix
`List<Animal> l = new ArrayList<Dog>();`	`incompatible types: ArrayList<Dog> cannot be converted to List<Animal>`	Generics are not covariant.

Table 4.4.16: Common compile errors related to polymorphism.

Mistake	Result	Fix
An unchecked downcast: `(Dog) h`, although `h` may be a cat	`ClassCastException`	Test with `instanceof` first.
Thinking "a cast changes the object"	`((Animal) dog).makeSound()` still prints `Woof woof!`	A cast changes only the type that is looked at.
Thinking fields are overridden too	`h.kind` is read by the reference type: `mammal`	Make the field `private` and override the getter method.
Mistaking overloading for overriding: `equals(Person k)`	Collections do not call this method; `Object.equals` runs	The signature becomes `equals(Object)`; `@Override` catches the mistake.
Comparing objects with `==`	References are compared, not contents	The overridden `equals` is used.
`instanceof` chains everywhere	Silently wrong results as new subclasses are added	The behavior is moved to an overridden method.
Calling an overridable method from a constructor	`null` output or `NullPointerException`	Only `private`, `final` or `static` methods are called from a constructor.
Relying on array covariance: `Animal[] d = new Dog[2]; d[1] = new Cat();`	`ArrayStoreException: Cat`	The array is created with the superclass type.
Thinking boxing comes first	`f(5)` → `f(long)`, not `f(Integer)`	Widening is in phase 1.
Extending `equals` in a subclass in one direction only	`p.equals(k)` `true`, `k.equals(p)` `false`	A `getClass()` comparison, or composition.

Table 4.4.17: Common run-time and logic errors related to polymorphism.

The error `equals(Person k)` in the table needs a separate explanation. The signature of the method in `Object` is `equals(Object)`. Writing `equals(Person k)` in a subclass does not override it; it adds a new overload with the same name. In the call `a.equals(b)` made with two variables of type `Person`, the new method is chosen and seems to work correctly. But `ArrayList.contains` or `HashSet` pass the argument with type `Object`. The compiler sees only `equals(Object)` in that code; since `Person` does not override it, the reference comparison of `Object` runs. Had `@Override` been written on the method, the compiler would say `method does not override or implement a method from a supertype` and the mistake would have been caught from the start.

4.4.9 Summary

- Polymorphism means that the same expression leads to different behavior in different objects. The code is written once with the superclass type; each object answers the call with its own version.

- Compile-time polymorphism is provided by overloading, run-time polymorphism by overriding and dynamic binding. The compiler chooses the first, the JVM the second.
- Every reference variable has a declared type and the actual type of the object it points to. The declared type decides which methods can be called and which assignments are valid; the actual type decides which version runs.
- The reference rule: a superclass reference can point to a subclass object; the reverse does not compile. `Animal h = new Dog();` is valid, `Dog k = new Animal();` is an error.
- Upcasting is automatic and always safe. A downcast is written explicitly; if the object does not fit, `ClassCastException` is thrown at run time. A cast between sibling classes never compiles.
- The dot operator is applied before the cast: the correct form is `((Dog) h).bark()`.
- `instanceof` tests the type of the object before the cast. Since Java 16, pattern matching combines the test and the cast: `if (x instanceof Dog kk)`.
- Arrays are covariant; writing an element of the wrong type gives `ArrayStoreException`. Generics are not covariant and prevent the same error at compile time.
- Method lookup starts at the actual class of the object and climbs up. The JVM does this with method tables kept per class, with a single array access; only the method of the reference type is written in the bytecode (`invokevirtual`).
- Fields and `static`, `private` and `final` methods are not bound dynamically. A field with the same name or a `static` method with the same signature in a subclass hides the one in the superclass; it does not override it.
- An overridable method called from a constructor runs before the subclass's fields are assigned. Only `private`, `final` or `static` methods are called from a constructor.
- The overloading choice is made at compile time by the declared types of the arguments. If several versions fit, the most specific is chosen; if there is no single most specific version, the call is ambiguous.
- Overloading is resolved in three phases: without boxing (widening), boxing, variable arity. If a candidate is found in a phase, later ones are not reached; narrowing is never tried.
- Polymorphic code is open for extension and closed for modification: adding a new subclass changes no existing loops or methods. Parameter and variable types are chosen as the most general type that is enough to do the job.
- If `equals` is extended in a subclass in one direction only, symmetry breaks. The solution is a `getClass()` comparison or composition instead of inheritance.

4.4.10 Key terms

polymorphism compile time / run time declared type / actual type reference superclass / subclass is-a / has-a relationship `Object` class upcasting / downcasting cast (explicit conversion) widening / narrowing conversion dynamic binding virtual method table method overriding

method overloading method signature autoboxing / unboxing constructor stack trace generics factory method composition abstract method hiding array covariance pattern matching open/closed principle

---

4.4.11 Test Yourself

The questions use the example hierarchy: `Dog` overrides `makeSound()` and `run()`; `Cat` overrides only `makeSound()`.

1. What does `Animal h = new Cat(); h.run(); h.makeSound();` print?

   Answer

   `Animal runs` and `Meow!`. For `run()` the search starts at `Cat`; since `Cat` does not override this method, it ends at `Animal`. `makeSound()` is found in `Cat`.

2. After `Object o = new Cat();`, what is the result for each of the lines `Animal h = (Animal) o;`, `Cat c = (Cat) o;`, `Dog k = (Dog) o;` and `Cat c2 = o;`?

   Answer

   The first three lines compile; a downcast from `Object` to any class can be written. At run time the first two succeed, because the object is a cat and a cat is an animal. The third gives `ClassCastException`. The fourth line does not compile at all: there is no explicit cast (`incompatible types: Object cannot be converted to Cat`).

3. Given `class U { int x = 1; int getX() { return x; } }` and `class V extends U { int x = 2; int getX() { return x; } }`. What does `u.x + " " + u.getX() + " " + ((V) u).x` give for `U u = new V();`?

   Answer

   `1 2 2`. `u.x` is a field access and is read by the declared type (`U`): 1. `u.getX()` is bound dynamically; the `x` inside `V.getX()` is the field of `V`: 2. After the cast the declared type is `V`: 2.

4. Let `static void g(Object o)` print `"O"` and `static void g(String s)` print `"S"`. For `Object`

`a = "x";`, what do `g(a)`, `g((String) a)` and `g(null)` print?

Answer

`O`, `S`, `S`. The declared type of `a` is `Object`; that the object is a string does not matter. With the cast the type becomes `String`. `null` fits both versions; but `String` is a subtype of `Object` and the two candidates lie on a single chain. The more specific `g(String)` is chosen; there is no ambiguity.

5. With `static void h(double d)` and `static void h(Integer i)` present, which version do `h(5)`, `h(Integer.valueOf(5))` and `h(5L)` choose?

Answer

`h(5)`: in phase 1 the widening `int` → `double` fits; `Integer` needs boxing and is not tried: `double`. `h(Integer.valueOf(5))`: in phase 1 the exact match is `h(Integer)`; reaching `double` needs unboxing: `Integer`. `h(5L)`: `long` widens to `double`: `double`.

6. Given `class A { A() { m(); } void m() { System.out.println("A"); } }` and `class B extends A { int n = 7; void m() { System.out.println("B " + n); } }`. What does `new B();` print?

Answer

`B 0`. The call `m()` inside `A()` is bound dynamically, and since the object is a `B`, `B.m()` runs. At that moment `n` has not yet been given its initial value; the default value of `int` is 0. `n = 7` runs after `A()` returns.

7. Which of the lines `Object[] d = new Integer[2]; d[0] = 5; d[1] = "a";` causes a problem?

Answer

All three lines compile. `d[0] = 5` works: 5 is boxed to `Integer` and fits the actual element type of the array. `d[1] = "a"` gives `ArrayStoreException: java.lang.String` at run time; the array is really an `Integer[]`.

8. Given `class P { String name() { return "P"; } String print() { return name(); } }` and `class Q extends P { String name() { return "Q"; } String print() { return "[" +`

`super.print() + "]"; } }`. What does `p.print()` return for `P p = new Q();`?

Answer

`[Q]`. `p.print()` is bound dynamically: `Q.print()`. `super.print()` calls `P.print()` directly (`invokespecial`). The `name()` inside it is `this.name()`, and `this` is still a `Q` object: `Q.name()` → `"Q"`. `super` fixes only the target of that one call; the calls inside are still bound dynamically.

4.4.12 Further reading

- Oracle, *The Java Tutorials*, the "Polymorphism" and "Overriding and Hiding Methods" pages: explain dynamic binding and the hiding of fields and `static` methods with short examples. It is Java's official, free starting point.
- J. Bloch, *Effective Java* (3rd ed.), Item 10 "Obey the general contract when overriding equals" and Item 19 "Design and document for inheritance or else prohibit it": discuss in detail the `equals` symmetry trap and the danger of calling an overridable method from a constructor.
- P. Deitel and H. Deitel, *Java: How to Program* (11th ed.), Chapter 10 "Object-Oriented Programming: Polymorphism and Interfaces": builds up polymorphism step by step with a payroll example; it also moves on to abstract classes and interfaces.
- J. Gosling et al., *The Java Language Specification, Java SE 21 Edition*, 5.5 "Casting Contexts" and 15.12 "Method Invocation Expressions": the exact definition of casts and of the three phases of overload resolution.

4.4.13 References

1. Oracle, *The Java Tutorials*: "Polymorphism"; "Overriding and Hiding Methods"; "Inheritance".
2. J. Gosling et al., *The Java Language Specification, Java SE 21 Edition*, Oracle, 2023 (5.5 Casting Contexts; 8.4.8 Inheritance, Overriding, and Hiding; 15.12 Method Invocation Expressions).
3. P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson (Chapter 10: Object-Oriented Programming: Polymorphism and Interfaces).
4. K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly, 2022 (Inheritance and Polymorphism).
5. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson, 2022 (Chapter 5: Inheritance).
6. A. B. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019.
7. J. Bloch, *Effective Java*, 3rd ed., Addison-Wesley, 2018.

PART II · CHAPTER 4.5

Abstract Classes and Interfaces

Inheritance gathers shared code in the superclass, and polymorphism makes the same call run the method of each object's own class. These two tools leave one gap: a superclass sometimes cannot write the body of a method. The area of a general “shape” cannot be computed; only a square, a circle or a triangle knows the formula. Java offers two tools for this situation: the **abstract class** and the **interface**. Both say “what to do” and leave “how to do it” to the subtypes. These notes first cover the abstract class and the abstract method, then the interface, the methods with bodies in interfaces, and sealed types. Lambda expressions, the short notation for interfaces with a single method, generic classes with type parameters, and annotations that add metadata to code are among the topics too. The last section explains five design patterns, most of them built on interfaces (Singleton, Factory, Observer, Strategy and MVC), with small console programs. All programs were compiled and run with JDK 21; the outputs, the final values in the trace tables and the compiler messages come from these runs.

LEARNING OUTCOMES

Abstract class

Define `abstract` classes and methods; explain why an object cannot be created from an abstract class and which methods a concrete subclass must write.

Template method

Show, by tracing the code step by step, that a concrete method of an abstract class calls an abstract method written in the subclass.

Interface

Define an `interface`; implement one or several interfaces with `implements`; use `default`, `static` and `private` interface methods.

Lambda

Recognize a functional interface; write a one-method behavior as a lambda expression and trace when the lambda runs.

Design decision

Recognize sealed types; decide whether a problem needs an interface or an abstract class (or both); use generic types and annotations.

Design patterns

Explain which problem the Singleton, Factory, Observer, Strategy and MVC patterns solve and how they rely on interfaces.

4.5.1 Abstraction

Someone who drives a car knows the steering wheel, the gear stick and the pedals. They do not have to know how the fuel burns inside the engine or how the gearbox changes its gears. The face the car presents to the driver is simple; the complex detail stays behind that face. Moreover, the same face is the same on different cars. Someone who can drive a gasoline car can drive an electric car too, because both obey the contract of steering wheel and pedals.

Definition 4.5.1

Abstraction is hiding complex details and showing only the essential features. In programming, abstraction separates "what" a job does from "how" it does it.

The same separation is useful in a program. Suppose a drawing program wants to compute the total area of the shapes on the screen. Every shape has an area; this is the essential property of the concept "shape". But the area of a general shape cannot be computed. The formula is $a \cdot a$ for a square, $\pi \cdot r^2$ for a circle, $b \cdot h / 2$ for a triangle. Only the concrete shape knows the formula. Figure 4.5.1 draws this situation: the common concept is at the top, the detail at the bottom.

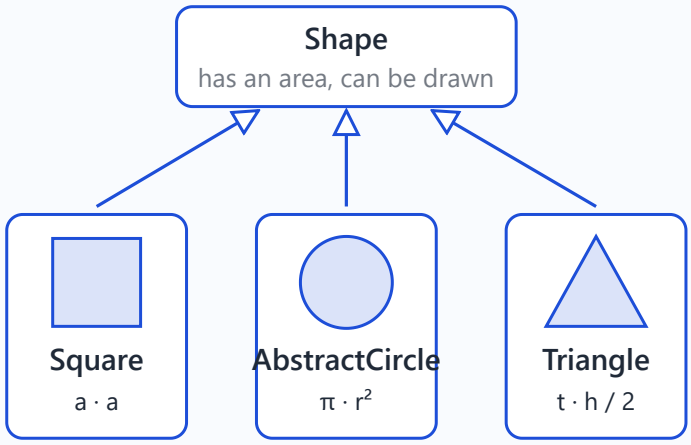


Figure 4.5.1: "Shape" is a concept: it has an area but the area cannot be computed. Square, Circle and Triangle make the concept concrete, and each computes the area with its own formula.

The code that computes the total area should rely only on the knowledge that “every shape has an `area()` method”. It does not ask whether a shape is a square or a circle; it asks each shape for its area. If a hexagon is added to the program tomorrow, the summing code does not change; only the new class is written. This is the gain of abstraction: the code that uses a type depends on the concept, not on the detail.

Java has two tools for abstraction: **abstract classes** (`abstract class`) and **interfaces** (`interface`). Both say what is to be done and leave how to do it to the subclasses. The difference between them in brief is this. An abstract class is the common ancestor of closely related classes; it can have fields, a constructor and ready-made methods. An interface is a contract that gives a common ability to classes that are not related to each other.

4.5.2 Abstract classes

In inheritance, the superclass carries the shared fields and methods of the subclasses. Class `Shape` can write the shared `name` field and the `move` method of all shapes. It gets stuck at the `area()` method. What should the body be? Writing `return 0;` is wrong. No shape's area is really 0. If a subclass forgets to override the method, the program silently produces a wrong result. The right solution is not to write the body at all and to tell the compiler so explicitly.

4.5.2.1 Defining an abstract class

> **Definition 4.5.2**
>
> An **abstract class** is a class declared with the `abstract` keyword from which no object can be created. An abstract class can contain both **abstract methods** (without a body) and **concrete methods** (with a body). A subclass that writes all the abstract methods is called a **concrete class**; objects can be created only from such a class.

```
public abstract class Shape {
    // concrete (non-abstract) method
    void move(int x, int y) {
        System.out.println("Moved to (" + x + ", " + y + ")");
    }
    // abstract method (subclasses must implement)
    abstract void draw();
}
```

This `Shape` class has two methods. `move` is concrete: its body is ready. Subclasses inherit it as it is, or override it if they wish. Since moving a shape to a point requires the same work for every shape, writing the body once in the superclass is right. `draw` is abstract: it has only a signature. Drawing a

square and drawing a circle are different jobs; so every concrete subclass must write its own `draw()` body.

An abstract class is like an unfinished plan. Some rooms are finished, for others only the place is fixed. The architect who takes over the plan fills the empty places without touching the finished rooms. The small program below completes the class with a `Square` and calls both methods:

```
abstract class Shape {
    // concrete (non-abstract) method
    void move(int x, int y) {
        System.out.println("Moved to (" + x + ", " + y + ")");
    }
    // abstract method (subclasses must implement)
    abstract void draw();
}
class Square extends Shape {
    void draw() { System.out.println("Drawing a square"); }
}
public class D03 {
    public static void main(String[] args) {
        Shape s = new Square();
        s.move(3, 4);
        s.draw();
    }
}
```

```
Moved to (3, 4)
Drawing a square
```

`Square` has written only `draw()`; it gets `move` from `Shape`. The type of the variable is the abstract `Shape`, but the object it points to is a `Square`. In the call `s.draw()`, which body runs is chosen at run time by the class of the object, as in polymorphism. This decision is called **dynamic binding**.

The other properties of an abstract class are the same as those of an ordinary class. It can have fields, constructors and `static` members. Other classes extend it with `extends`. Figure 4.5.2 draws a `Shape` class with four methods and the three concrete classes that extend it. The drawing is in UML (Unified Modeling Language) notation.

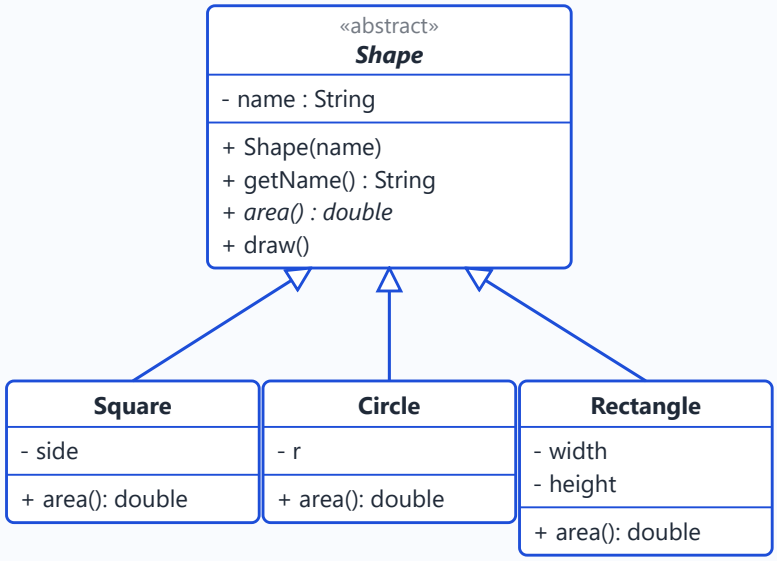


Figure 4.5.2: In UML the name of an abstract class and its abstract methods are written in italics. The `name` field, the constructor and the `getName()` method of `Shape` are concrete; `area()` and `draw()` are abstract. `Square`, `Circle` and `Rectangle` carry their own fields and write `area()` with their own formulas.

The gain of the design in the figure is that the shared state (the fields) and behavior are gathered in a single place. The `name` field and the `getName()` that reads it are not written separately in three classes. The subclasses fill in the abstract methods according to their needs. That is why abstract classes often sit at the top of an inheritance hierarchy.

4.5.2.2 No object can be created from an abstract class

The most striking rule about an abstract class is that no object can be created from it with `new`. Of the three lines below, the first does not compile and the other two do:

```
Shape s1 = new Shape("?");     // ERROR
Shape s2 = new Square(3);        // OK: the variable type may be abstract
Shape[] arr = new Shape[3];   // OK: the array holds only references
```

```
AbstractNew.java:8: error: Shape is abstract; cannot be instantiated
        Shape s1 = new Shape("?");
                   ^
1 error
```

The ban is not arbitrary. The `area()` method of `Shape` has no body. If `new Shape("?")` were allowed, there would be no code to run in the call `s1.area()`. The compiler prevents this up front: an object with a missing body is never created.

The name of an abstract class, however, can be used freely as a reference type. A variable, a parameter, an array element and a return type can be `Shape`. The `Shape s2 = new Square(3);` on the second line is exactly the notation abstraction wants: the type of the variable is the concept, the class of the object is the detail. The third line often surprises students. `new Shape[3]` does not create a `Shape` object; it creates an array with three slots, and each slot starts as `null`. The array object holds references, not objects of the abstract class. This is clearly seen when the lines are run:

```
abstract class Shape {
    private String name;
    Shape(String name) { this.name = name; }
    abstract double area();
}
class Square extends Shape {
    private double side;
    Square(double side) { super("Square"); this.side = side; }
    double area() { return side * side; }
}
public class D05 {
    public static void main(String[] args) {
        Shape s2 = new Square(3);        // OK: the variable type may be abstract
        Shape[] arr = new Shape[3];   // OK: the array holds only references
        System.out.println(s2.area());
        System.out.println(arr.length + " " + arr[0]);
    }
}
```

```
9.0
3 null
```

A class cannot be both `abstract` and `final`. `final` forbids extending the class; `abstract` requires it to be extended, because an object can be created only from a subclass. The two modifiers contradict each other:

```
abstract final class Shape { }
public class AbstractFinal { }
```

```
AbstractFinal.java:1: error: illegal combination of modifiers: abstract and final
abstract final class Shape { }
               ^
1 error
```

4.5.2.3 Abstract methods and their rules

> **Definition 4.5.3**
>
> An **abstract method** is a method defined without a body, with only its signature. It contains no braces and ends with a semicolon. Every concrete subclass must implement it, that is, write its body.

```
public abstract void draw();
```

A class that has an abstract method must itself be abstract. In the two files below, `Shape` declares the abstract method `draw()` and `Circle` implements it:

```
public abstract class Shape {
    // abstract method
    public abstract void draw();
}

// in a separate file:
public class Circle extends Shape {
    // must implement draw
    public void draw() {
        System.out.println("Drawing a circle");
    }
}
```

```
Drawing a circle
```

The call `new Circle().draw();` runs the body that `Circle` wrote. Implementing an abstract method follows the same rules as overriding an ordinary method. The signature (name and parameter list) must be exactly the same, and access must not be narrowed. Since the method in the superclass is `public`, the one in `Circle` is `public` too.

The rules about abstract methods fit in a short table. Table 4.5.1 gives correct and wrong forms together with real compiler messages.

Written	Result
`abstract void draw();` (in an abstract class)	Correct
`abstract void draw() { }`	`abstract methods cannot have a body`
`class Shape { abstract void draw(); }`	`Shape t not abstract and does not override abstract method draw() in Shape`

Written	Result
`class Circle extends Shape { }` (no `draw`)	`Circle t not abstract and does not override abstract method draw() in Shape`
`private abstract`, `static abstract`, `final abstract`	`illegal combination of modifiers: abstract and private` (the same message for `static` and `final`)
`abstract class Circle extends Shape { }`	Correct: an abstract subclass can leave the implementation to its own subclasses
`abstract class A { void f() { } }`	Correct: an abstract class with no abstract methods can also be written; still, `new A()` cannot be done

Table 4.5.1: Rules for abstract methods and the javac messages.

The message in the third row of the table looks odd at first sight: "`Shape` is not abstract and does not override abstract method `draw()` in `Shape`." The compiler gives every class with an abstract method two options: either the class becomes abstract or it writes the method. A class not writing the method it declared itself produces the same message. The fourth row is the same rule in the subclass.

Why are the three modifiers in the fifth row forbidden? All three prevent the method from being overridden in a subclass. A `private` method is invisible to the subclass. A `static` method is tied to the class, not to the object, and does not take part in dynamic binding. A `final` method cannot be overridden. An abstract method exists only to be overridden. The `static` version of the message looks like this:

```
abstract class Shape {
    static abstract void draw();
}
public class StaticAbstract { }
```

```
StaticAbstract.java:2: error: illegal combination of modifiers: abstract and static
    static abstract void draw();
                         ^
1 error
```

The last two rows show two allowed but little-known cases. A subclass of an abstract class can be abstract too; then the obligation to write the method passes to a lower generation. A class with no abstract method at all can also be marked `abstract`. This is the way of saying "do not create objects directly from this class, only extend it". In the program below both cases compile and run:

```
abstract class Shape { abstract void draw(); }
abstract class Circle extends Shape { }
abstract class A { void f() { System.out.println("f"); } }
class Circle extends Circle { void draw() { System.out.println("Drawing a circle"); } }
public class Main {
    public static void main(String[] args) {
        Shape s = new Circle();
        s.draw();
        A a = new A() { };   // anonymous concrete subclass
        a.f();
    }
}
```

```
Drawing a circle
f
```

Since `AbstractCircle` stays abstract, it has not written `draw()`; the concrete `Circle` that extends it writes it. The expression `new A() { }` creates not an object of `A` but an object of an anonymous subclass that extends `A`. This notation is called an anonymous class.

Common mistake

Writing the signature of an abstract method wrongly. If the abstract method in `Shape` is `double area()` and the subclass writes `double Area()` or `double area(int x)`, it is a new method. The abstract `area()` is still missing and the compiler rejects the class:

```
WrongSignature.java:2: error: Square is not abstract and does not override abstract
method area() in Shape
class Square extends Shape {
^
1 error
```

The message shows not the place of the mistake but its consequence; the typo may go unnoticed. If `@Override` is written above the method, the compiler states the real error directly: `method does not override or implement a method from a supertype`.

4.5.2.4 The constructor of an abstract class

If no object can be created from an abstract class, what is its constructor for? The answer lies in how a subclass object is built. When a `Square` object is created, the part of the object that comes from `Shape` must be built too: the `name` field must be given a value. The constructor of `Shape` does this

job and the subclass calls it with `super(...)`. In the program below the `Shape` constructor also prints a message, so the output shows when the call is made.

```
abstract class Shape {
    private String name;
    Shape(String name) {
        this.name = name;
        System.out.println(name + " created");
    }
    abstract double area();
}
class Square extends Shape {
    private double side;
    Square(double side) {
        super("Square");
        this.side = side;
    }
    double area() { return side * side; }
}
// in main:
Shape s = new Square(3);
System.out.println(s.area());
```

```
Square created
9.0
```

Table 4.5.2 gives the ten steps of the program. The line numbers are the lines of the code above. The "Call stack" column lists the frames on the stack at that moment from left to right, from the bottom to the top.

Step	Line	Call stack	Variables	New output
1	18: `Shape s = new Square(3);`	main()	s=—	—
2	11: `Square(double side) {`	main() → Square(3)	side=3.0	—
3	12: `super("Square");`	main() → Square(3)	side=3.0	—
4	3: `Shape(String name) {`	main() → Square(3) → Shape("Square")	name="Square"	—
5	4: `this.name = name;`	main() → Square(3) → Shape("Square")	name="Square"	—
6	5: `System.out.println(name + " created");`	main() → Square(3) → Shape("Square")	name="Square"	`Square created`
7	13: `this.side = side;`	main() → Square(3)	side=3.0	—
8	18: `Shape s = new Square(3);`	main()	s=→ Square	—

Step	Line	Call stack	Variables	New output
9	15: `double area() { return side * side; }`	main() → Square.area()	side=3.0	—
10	19: `System.out.println(s.area());`	main()	s=→ Square	`9.0`

Table 4.5.2: Trace of the expression `new Square(3)` and the call `s.area()`.

In the first step, `new Square(3)` reserves space for a `Square` object on the heap. The object has two fields: `name` from `Shape` and `side`, which belongs to `Square`. Both get their default values: `null` and `0.0`. In the second step the constructor of `Square` starts to run. The argument is the `int` 3 and the parameter is a `double`; the value is widened automatically to 3.0.

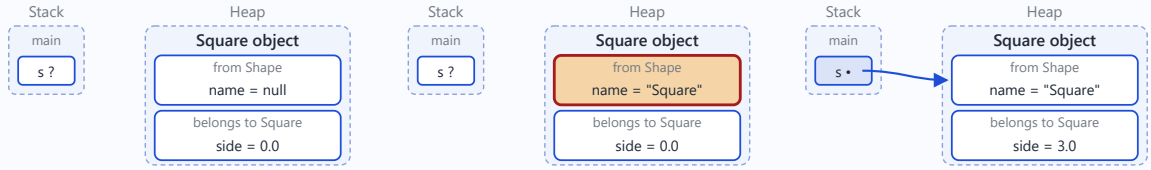


Figure 4.5.3: Three moments. On the left `new` has allocated the object, and the fields hold default values. In the middle the `Shape` constructor has built the part of the object that comes from `Shape`: `name = "Square"`. On the right the `Square` constructor is finished too and the reference has been assigned to `s`.

In the third step the first line of the `Square` constructor, `super("Square")`, runs and the superclass constructor is called. In the fourth step a `Shape("Square")` frame is added to the stack. There is a very important detail here: a separate `Shape` object is **not created**. There is still a single object on the heap. The `Shape` constructor builds the part of the same object that comes from `Shape`. In the fifth step `this.name = name` sets the field of this part to "Square", in the sixth step the message is printed and the frame of `Shape` leaves the stack.

In the seventh step control returns to the constructor of `Square` and `this.side = side` runs. The object is now completely built. In the eighth step the reference that is the value of the `new` expression is assigned to `s`. The type of the variable is the abstract `Shape`, and the object it points to is a `Square`. The drawing on the right of Figure 4.5.3 shows this moment.

The ninth step does the real work of abstraction. When `s.area()` is compiled, the compiler looks only at `Shape`: `Shape` has a method called `area()` that returns `double`, so the call is valid. At run time the JVM looks at the actual class of the object and runs `Square.area()`: 3.0 · 3.0 = 9.0. In the tenth step this value is printed.

The order of the constructor chain is the same as in inheritance: first the superclass constructor, then the subclass's. The abstract class does not change this order; it only forbids creating an object directly from the top link of the chain. Since `Shape` has no no-argument constructor, the constructor of `Square` must write `super("Square")`. Had it not, the compiler would insert `super()` itself and report an error because it cannot find the no-argument constructor.

4.5.2.5 The template method

A concrete method of an abstract class can call an abstract method of the same class. At first hearing this seems contradictory: how can a method without a body be called? The call compiles, because the signature of the method is known. At run time `this` always points to an object of a concrete subclass, and in that object's class the method has a body. The `Beverage` class below fixes the order in which a beverage is prepared:

```
abstract class Beverage {
    final void prepare() {
        boilWater();
        brew();
        System.out.println("Served");
    }
    void boilWater() {
        System.out.println("Water boiled");
    }
    abstract void brew();
}
class Tea extends Beverage {
    void brew() {
        System.out.println("Tea brewed");
    }
}
// in main:
Beverage i = new Tea();
i.prepare();
```

```
Water boiled
Tea brewed
Served
```

`prepare()` calls three steps in order: boiling the water, brewing, serving. Boiling water is the same for every beverage; `boilWater()` is written concretely in `Beverage`. Brewing differs by beverage; `brew()` is abstract and `Tea` fills it in with its own body. `prepare()` is `final`: subclasses cannot override it, that is, they cannot disturb the order of preparation.

Step	Line	Call stack	Variables	New output
1	18: `Beverage i = new Tea();`	main()	i=→ Tea	—
2	2: `final void prepare() {`	main() → prepare()	this=→ Tea	—
3	3: `boilWater();`	main() → prepare()	this=→ Tea	—
4	8: `System.out.println("Water boiled");`	main() → prepare() → boilWater()	this=→ Tea	`Water boiled`
5	4: `brew();`	main() → prepare()	this=→ Tea	—
6	14: `System.out.println("Tea brewed");`	main() → prepare() → Tea.brew()	this=→ Tea	`Tea brewed`

Step	Line	Call stack	Variables	New output
7	5: `System.out.println("Served");`	main() → prepare()	this=→ Tea	`Served`
8	19: `i.prepare();`	main()	i=→ Tea	—

Table 4.5.3: Trace of the call `i.prepare()`. `this` points to the `Tea` object on which the method was called.

Table 4.5.3 gives the call step by step. In the first step the type of the variable is the abstract `Beverage` and its object is a `Tea`. `Tea` is concrete because it implements the only abstract method `brew()`, and its object can be created. In the second step `i.prepare()` is called. `Tea` has no `prepare()`; the inherited version from `Beverage` runs. In the third and fourth steps `boilWater()`, that is `this.boilWater()`, is called, and the concrete body in `Beverage` prints "Water boiled".

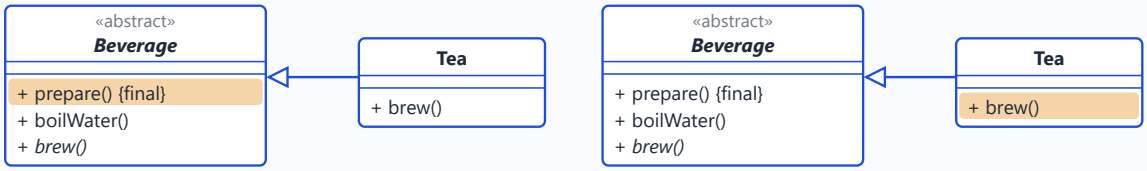


Figure 4.5.4: The moment on the left: inside `prepare()` the call to `brew()` has been reached; `Beverage` has no body for this method. The moment on the right: since `this` is a `Tea` object, the call goes to `Tea.brew()`.

The fifth step is the critical moment. `prepare()` calls `brew()`. `Beverage` has no body for this method, but `this` is a `Tea` object. Dynamic binding directs the call to `Tea.brew()` and in the sixth step "Tea brewed" is printed (Figure 4.5.4). The code in the abstract class has called the code of a subclass that did not yet exist when the abstract class was written. In the seventh step the serving message is printed.

Definition 4.5.4

The **template method** is a design that defines the skeleton of an algorithm in the superclass and leaves the varying steps to the subclasses. The method that builds the skeleton is usually `final`; the varying steps are abstract methods.

The strength of the template method shows when a new beverage is added. A `Coffee` class writes only `brew()`; it takes the water-boiling and serving steps, together with their order, ready-made:

```
abstract class Beverage {
    final void prepare() {
        boilWater();
        brew();
        System.out.println("Served");
    }
    void boilWater() {
        System.out.println("Water boiled");
    }
    abstract void brew();
}
class Tea extends Beverage {
    void brew() {
        System.out.println("Tea brewed");
    }
}
class Coffee extends Beverage {
    void brew() {
        System.out.println("Coffee filtered");
    }
}
public class Main {
    public static void main(String[] args) {
        Beverage[] order = { new Tea(), new Coffee() };
        for (Beverage i : order) {
            i.prepare();
        }
    }
}
```

```
Water boiled
Tea brewed
Served
Water boiled
Coffee filtered
Served
```

The loop calls the same `prepare()` on both objects. The skeleton does not change; the middle line runs the method of each object's own class. Since the order of preparation is written in a single place, adding a “warm the cup” step before serving, for example, changes all beverages together.

Common mistake

Trying to override a `final` template method in a subclass. The template method is written `final` to protect the skeleton. If a subclass wants to rewrite it, the compiler rejects it:

```
FinalPrepare.java:7: error: prepare() in Tea cannot override prepare() in Beverage
  void prepare() { }
       ^
  overridden method is final
1 error
```

If part of the order really must be variable, the template is not broken. A "hook" method with a body that can be overridden is added to the skeleton (for example `boolean addMilk() { return false; }`).

4.5.3 Interfaces

An abstract class works well as the common ancestor of related classes. A square and a circle are both shapes. But some things in common do not come from kinship. A bird, an airplane and a drone can fly; the three have no common ancestor, and should not have one. A duck both flies and swims; since it has a single superclass, it cannot derive from two separate abstract classes at the same time. Java offers a second tool of abstraction for this need: the interface.

4.5.3.1 An interface is a contract

Definition 4.5.5

An **interface** is a type that specifies the methods classes must implement. It is defined with the `interface` keyword. A class **implements** an interface with `implements`, that is, it writes the bodies of the abstract methods in the interface.

An interface is a **contract**. It guarantees that every class implementing it will offer certain methods. It is the meeting point between two pieces of software: the using side knows only the interface and does not know the class behind the contract. Figure 4.5.5 draws this relationship.

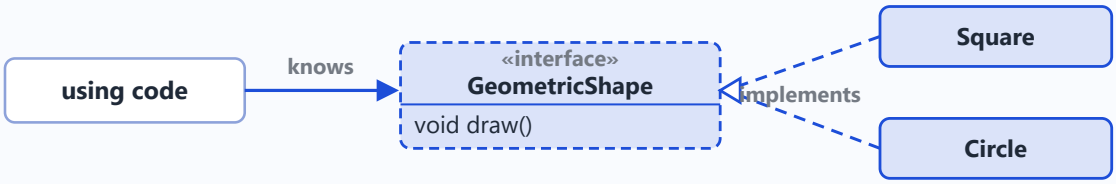


Figure 4.5.5: The using code knows only the `GeometricShape` interface and its `draw()` method. `Square` and `Circle` implement the same contract (dashed arrows).

An everyday analogy describes the interface well: the socket and the plug. The socket is a contract. Whatever device is plugged in, it gives the same voltage. Both an iron and a computer obey the standard of the socket. The grid does not know the inside of the device, and the device does not know how the grid is built. Making a new device does not require changing the grid; following the standard is enough.

The rules of an interface differ from those of a class:

- Methods in an interface are `public`, whether or not `public` is written.
- A method that is not `static`, `default` or `private` is `abstract` by itself.
- Only `public static final` constants can be fields. There are no instance fields and no constructors.

The difference between an abstract class and an interface can be summed up with two relationship names. An abstract class establishes an “is-a” relationship: a square is a shape. An interface defines a “can-do” ability: a duck can fly, a duck can swim.

4.5.3.2 Defining an interface

Below, on the left is the interface the programmer wrote, on the right how the compiler sees it. The compiler adds the modifiers that were not written:

```
public interface GeometricShape {
    // abstract method (public abstract)
    void draw();
    // constant (public static final)
    int SIDES = 4;
}
```

```
public interface GeometricShape {
    public abstract void draw();
    public static final int SIDES = 4;
}
```

The line `void draw();` is read as `public abstract void draw();`. The line `int SIDES = 4;` is a `public static final` constant. Constants are reached with the interface name: `GeometricShape.SIDES` gives 4. Constant names are traditionally written in `UPPER_CASE`. These implicit modifiers have three consequences, and the compiler checks all three.

First, giving a value to a constant in an interface is mandatory. A `final` field without a value cannot get one later either, because an interface has no constructor:

```
interface GeometricShape {
    int value;
}
public class FinalNoValue { }
```

```
FinalNoValue.java:2: error: = expected
    int value;
             ^
1 error
```

Second, the constant cannot be changed. A `final` field cannot be assigned a second time:

```
interface GeometricShape {
    void draw();
    int SIDES = 4;
}
public class AssignFinal {
    public static void main(String[] args) {
        GeometricShape.SIDES = 5;
    }
}
```

```
AssignFinal.java:7: error: cannot assign a value to static final variable SIDES
        GeometricShape.SIDES = 5;
                       ^
1 error
```

Third, an object cannot be created from an interface with `new` either. The message is the same as for an abstract class, because an interface is implicitly abstract too:

```
InterfaceNew.java:6: error: GeometricShape is abstract; cannot be instantiated
        GeometricShape g = new GeometricShape();
                           ^
1 error
```

An interface having no instance fields means it cannot hold state. An interface cannot say “it stores a counter value”; it can only say “it offers a method called `value()`”. Storing the value is the job of the implementing class. If `int value = 0;` is written in an interface, it is not a counter but an immutable constant; a method that tries to increase it does not compile:

```
interface Counter {
    int value = 0;
    default void inc() { value++; }
}
public class InterfaceField { }
```

```
InterfaceField.java:3: error: cannot assign a value to static final variable value
    default void inc() { value++; }
                         ^
1 error
```

4.5.3.3 Implementing an interface

A class implements an interface with the `implements` keyword. The class must write all the abstract methods of the interface or be `abstract` itself. The `Square` below writes `draw()`, the only abstract method of `GeometricShape`:

```
public class Square implements GeometricShape {
    // must implement draw
    public void draw() {
        System.out.println("Drawing a square");
    }
}
// in main:
GeometricShape g = new Square();
g.draw();
System.out.println(GeometricShape.SIDES);
```

```
Drawing a square
4
```

In the line `GeometricShape g = new Square();` the type of the variable is the interface. A variable of an interface type can point to an object of any class implementing that interface. When `g.draw()` is compiled, the compiler looks at the interface; at run time the method of `Square` runs. The last line reaches the constant by the interface name and prints 4.

The implementing method must be `public`. A method in an interface is implicitly `public` and an overriding method cannot narrow access. If the class writes only `void draw()`, the method gets package access and the compiler sees this as weakening access:

```
interface GeometricShape {
    void draw();
}
class Square implements GeometricShape {
    void draw() { System.out.println("Drawing a square"); }
}
public class InterfaceWeaker { }
```

```
InterfaceWeaker.java:5: error: draw() in Square cannot implement draw() in GeometricShape
    void draw() { System.out.println("Drawing a square"); }
         ^
  attempting to assign weaker access privileges; was public
1 error
```

If the method is not written at all, the message is the same as for an abstract class; the only difference is the last word, which says in which type the missing method is declared:

```
InterfaceMissing.java:4: error: Square is not abstract and does not override abstract
method draw() in GeometricShape
class Square implements GeometricShape { }
^
1 error
```

Common mistake

Forgetting to write `public` on an interface method. Since `public` is not written in the interface, one thinks it need not be written in the class either. But the modifier in the interface is implicit, the one in the class is not. If no modifier is written in the class, the method gets package access and the message `attempting to assign weaker access privileges; was public` appears. An interface method is always written `public` in the class.

4.5.3.4 One class, several interfaces

In Java a class can extend at most one superclass. In contrast, it can implement zero or more interfaces: `implements A, B, …`. If both are written, `extends` comes first, then `implements`. The `Duck` class below derives from `Animal` and takes on two abilities at once:

```
class Animal {
    String name;
    Animal(String name) { this.name = name; }
}
interface Flyable { void fly(); }
interface Swimmable { void swim(); }
class Duck extends Animal implements Flyable, Swimmable {
    Duck(String name) { super(name); }
    public void fly() { System.out.println(name + " is flying"); }
    public void swim() { System.out.println(name + " is swimming"); }
}
```

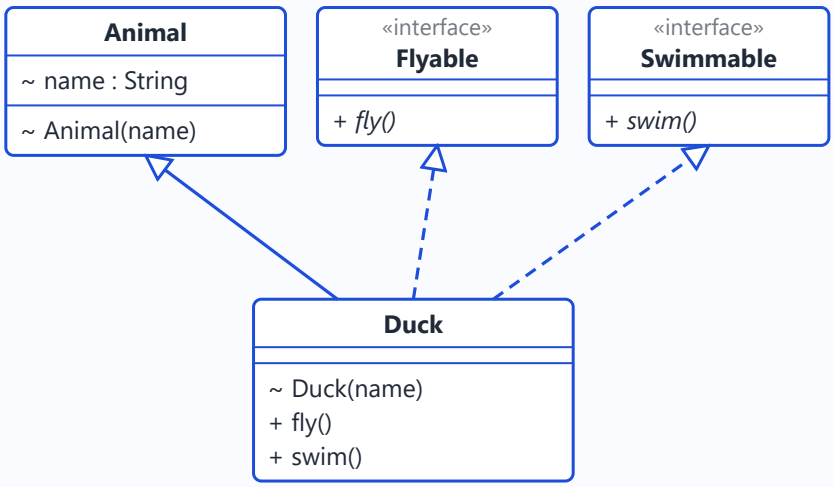


Figure 4.5.6: Class `Duck` extends `Animal` (solid line, hollow triangle) and implements the interfaces `Flyable` and `Swimmable` (dashed line).

The constructor of `Duck` calls the constructor of `Animal` with `super(name)` and gets the `name` field from it. The methods `fly()` and `swim()` fulfill the contracts of the two interfaces; both are `public`. Figure 4.5.6 shows these three relationships with the two kinds of UML line: inheritance is drawn solid, implementing an interface dashed.

Writing two superclasses is a syntax error. The compiler expects a single name after `extends`; when it sees the comma, it expects the class body to begin:

```
class A { }
class B { }
class C extends A, B { }
public class TwoExtends { }
```

```
TwoExtends.java:3: error: '{' expected
class C extends A, B { }
                   ^
1 error
```

A `Duck` object can now be seen from three different types: `Duck`, `Flyable` and `Swimmable`. The program below holds a single object with three different references:

```
Duck o = new Duck("Donald");
Flyable u = o;
Swimmable y = o;
u.fly();
y.swim();
// u.swim();   compile error!
System.out.println(o.name);
System.out.println(u == o);
```

```
Donald is flying
Donald is swimming
Donald
true
```

Step	Line	Variables	New output
1	1: `Duck o = new Duck("Donald");`	o=→ Duck	—
2	2: `Flyable u = o;`	o=→ Duck, u=→ Duck	—
3	3: `Swimmable y = o;`	o=→ Duck, u=→ Duck, y=→ Duck	—
4	4: `u.fly();`	o=→ Duck, u=→ Duck, y=→ Duck	`Donald is flying`
5	5: `y.swim();`	o=→ Duck, u=→ Duck, y=→ Duck	`Donald is swimming`
6	6: `// u.swim(); compile error!`	o=→ Duck, u=→ Duck, y=→ Duck	—
7	7: `System.out.println(o.name);`	o=→ Duck, u=→ Duck, y=→ Duck	`Donald`
8	8: `System.out.println(u == o);`	o=→ Duck, u=→ Duck, y=→ Duck	`true`

Table 4.5.4: Using the same `Duck` object through three references. The notation `→ Duck` means the variable points to the single `Duck` object on the heap.

The first three steps of Table 4.5.4 only assign. In the first step a single `Duck` object is created on the heap and `o` points to it. The `Flyable u = o;` in the second step does not copy the object; `u` points to the same object. Since a duck is a `Flyable`, this conversion is automatic, no explicit cast is needed. In the third step `y` points to the same object too. Now the single object is looked at from three different types (Figure 4.5.7).

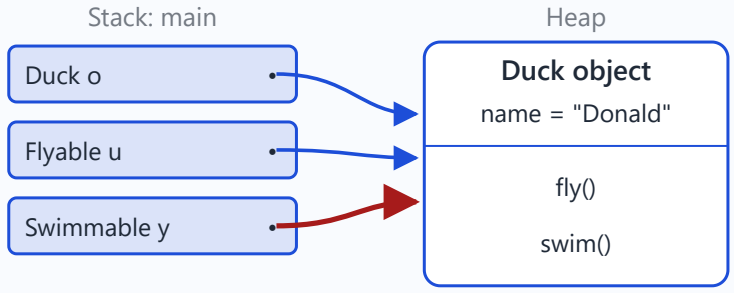


Figure 4.5.7: The variables `o`, `u` and `y` on the stack have different types, but all three point to the same `Duck` object on the heap.

In the fourth step `u.fly()` is called. The type of `u` is `Flyable`, and from this type only `fly()` can be called. What runs is the object's own `fly()` method: "Donald is flying". In the fifth step `y.swim()` runs; the type `Swimmable` knows `swim()`. The sixth step discusses the call in the comment line. Had `u.swim()` been written, the compiler would have rejected it:

```
RefType.java:10: error: cannot find symbol
        u.swim();
         ^
  symbol:   method swim()
  location: variable u of type Flyable
1 error
```

The object has a `swim()` method, but the compiler looks not at the object but at the type of the reference. There is no `swim()` in the `Flyable` contract. The type of the reference decides which methods can be called; the class of the object decides which body runs. These two rules are the heart of polymorphism. In the seventh step `o.name` is read; since the type of `o` is `Duck`, every member, fields included, can be reached. In the last step `u == o` compares the two references. Since both point to the same object, the result is `true`.

4.5.3.5 Inheritance between interfaces

An interface can extend another interface with `extends`. Multiple inheritance is allowed between interfaces: `interface C extends A, B` can be written. The extending interface inherits all the methods of the parent interface and adds new ones. The `SquareShape` below adds a `special()` to the `draw()` method of `GeometricShape`:

```java
interface GeometricShape {
    void draw();
    int SIDES = 4;
}
interface SquareShape extends GeometricShape {
    // may add methods or constants
    void special();
}
class SpecialSquare implements SquareShape {
    public void draw() {
        System.out.println("Drawing a special square");
    }
    public void special() {
        System.out.println("Diagonal: " + Math.sqrt(2));
    }
}
public class Main {
    public static void main(String[] args) {
        SquareShape k = new SpecialSquare();
        k.draw();
        k.special();
        GeometricShape g = k;   // can be assigned to the parent interface
        System.out.println(g instanceof SquareShape);
    }
}
```

```
Drawing a special square
Diagonal: 1.4142135623730951
true
```

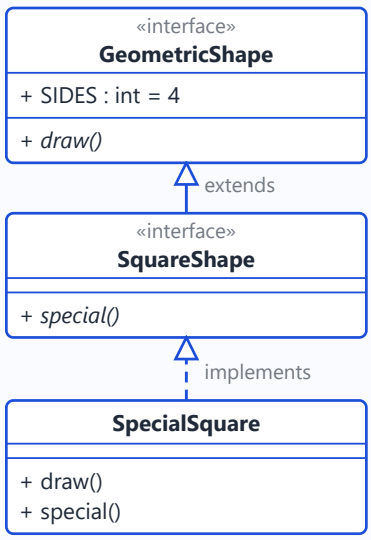


Figure 4.5.8: The `SquareShape` interface extends `GeometricShape`; the class `SpecialSquare` implements

`SquareShape` and writes both methods.

`SpecialSquare` declares that it implements only `SquareShape`, but the contract covers both methods. Had one been missing, the class would not compile. A `SpecialSquare` object is indirectly a `GeometricShape`; that is why `GeometricShape g = k;` compiles without a cast (Figure 4.5.8). The last line of the program is an `instanceof` test and prints `true`: the reference of the parent interface type still points to the same object. The 1.4142135623730951 on the Diagonal line is the `double` representation of $\sqrt{2}$, the diagonal of a square with side 1.

4.5.3.6 Programming to an interface type

The real strength of an interface is that it makes the using code independent of concrete classes. The way to do this is to write variable, parameter and return types as the interface instead of the concrete class. Consider the payment step of a shopping site. Payment can be made by card or by bank transfer; tomorrow cryptocurrency may be added too.

```
interface PaymentMethod { void pay(double amount); }
class CreditCard implements PaymentMethod {
    public void pay(double amount) {
        System.out.println("Card: " + amount + " TL");
    }
}
class Transfer implements PaymentMethod {
    public void pay(double amount) {
        System.out.println("Transfer: " + (amount + 5) + " TL");
    }
}
static void buy(PaymentMethod y, double total) {
    y.pay(total);
}
buy(new CreditCard(), 250.0);   // main
buy(new Transfer(), 120.0);
```

```
Card: 250.0 TL
Transfer: 125.0 TL
```

The method `buy(PaymentMethod y, double total)` does not know whether the payment is made by card or by transfer. It only says `y.pay(total)`; the object does the rest. Figure 4.5.9 draws this dependency: the only arrow out of `buy` goes to the interface.

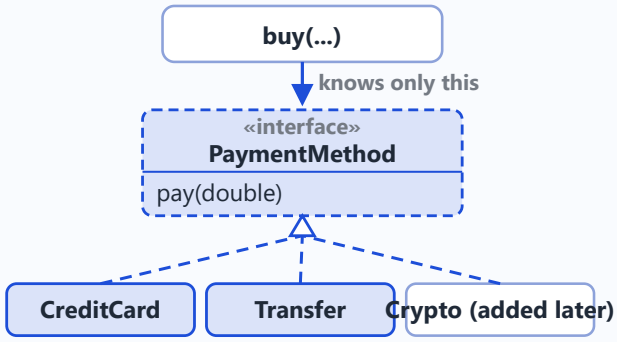


Figure 4.5.9: `buy` knows only the `PaymentMethod` interface. `CreditCard`, `Transfer` and the later-added `Crypto` implement the contract.

Step	Line	Variables	New output

1	15: `buy(new CreditCard(), 250.0); // main`
	12: `static void buy(PaymentMethod y, double total) {`
		y=→CreditCard #1, total=250.0	—

2	13: `y.pay(total);` 3: `public void pay(double amount) {` 4: `System.out.println("Card: " + amount + " TL");`	this=→CreditCard #1, amount=250.0	`Card: 250.0 TL`
3	15: `buy(new CreditCard(), 250.0); // main`	—	—
4	16: `buy(new Transfer(), 120.0);` 12: `static void buy(PaymentMethod y, double total) {`	y=→Transfer #2, total=120.0	—
5	13: `y.pay(total);` 8: `public void pay(double amount) {` 9: `System.out.println("Transfer: " + (amount + 5) + " TL");`	this=→Transfer #2, amount=120.0	`Transfer: 125.0 TL`
6	12: `static void buy(PaymentMethod y, double total) {` 13: `y.pay(total);`	—	—

Table 4.5.5: Trace of the method `buy` with two different payment objects. `#1` and `#2` are the serial numbers of the objects on the heap.

In the first step of Table 4.5.5, line 15 calls `buy` with a new `CreditCard` object. The type of the parameter `y` is the interface; the object that arrives is a `CreditCard`. In the second step the `y.pay(total)` on line 13 is bound dynamically. Since the object is a card, `CreditCard.pay()` runs and “Card: 250.0 TL” is printed. In the third step the call ends. No reference to the card object remains; the garbage collector may remove it.

In the fourth step the same `buy` is called, this time with a `Transfer` object. In the fifth step the same line 13 runs `Transfer.pay()` this time. The transfer adds a fee of 5 TL: 120.0 + 5 = 125.0. The conclusion of the sixth step is important: `buy` worked with two different classes but knows the name of neither. It will not change when a new payment type is added either.

This independence is called **loose coupling**. It has two concrete benefits. First, a new payment type is added just by writing a class that implements the interface; `buy` and the code that calls it do not change at all. Second, the code is easy to test. Instead of a real bank, a fake class that implements the interface is given; this class does not draw any money, it only records how much payment has reached it:

```
interface PaymentMethod { void pay(double amount); }
class Crypto implements PaymentMethod {
    public void pay(double amount) {
        System.out.println("Crypto: " + amount / 2000 + " units");
    }
}
class FakePayment implements PaymentMethod {
    double received = 0;
    public void pay(double amount) { received += amount; }
}
public class Main {
    static void buy(PaymentMethod y, double total) {
        y.pay(total);
    }
    public static void main(String[] args) {
        buy(new Crypto(), 500.0);
        FakePayment fake = new FakePayment();
        buy(fake, 99.5);
        buy(fake, 0.5);
        System.out.println(fake.received);
    }
}
```

```
Crypto: 0.25 units
100.0
```

`Crypto` is a payment type added later; it converts 500 TL into 0.25 units at a unit price of 2000 TL. `FakePayment` keeps the total of two payments: 99.5 + 0.5 = 100.0. That `buy` passes the right amount to the right object is thus tested without connecting to a real bank. Java's own library is written in the same fashion: in the line `List<String> l = new ArrayList<>();` the type of the variable is the interface `List`. If the program later wants to use `LinkedList` instead of `ArrayList`, only this line changes.

4.5.4 Lambda expressions

Most interfaces like `PaymentMethod` have a single abstract method. To implement such an interface one has to write a separate class, give it a name and place the body inside that class. If the class will be used only once, in one place, this ceremony is needlessly long. Lambda expressions, which came with

Java 8, let you write a one-method behavior directly where it will be used.

4.5.4.1 An anonymous function and its syntax

Definition 4.5.6

A **lambda expression** is an anonymous function made of a parameter list, an arrow (`->`) and a body: `(ParameterList) -> Expression`. An anonymous function represents the code itself directly, instead of a name.

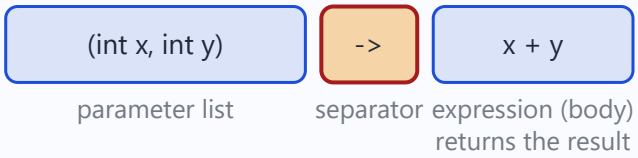


Figure 4.5.10: The three parts of a lambda expression: the parameter list in parentheses, the separating arrow and the body, which is an expression. The value of the expression is the result of the lambda.

The expression `(int x, int y) -> x + y` in Figure 4.5.10 takes two integers and returns their sum. The ordinary method that does the same job is:

```
static int add(int x, int y) {
    return x + y;
}
```

The differences between the two notations explain the short form of the lambda. A lambda has no name; the return type is not written, it is inferred from the type of the expression; in a single-expression body neither `return` nor braces are written. The syntax has a few forms:

- Parameter types are often not written; the compiler infers them from the context: `(x, y) -> x + y`.
- With a single parameter the parentheses are not needed either: `s -> s * 2`.
- A lambda with no parameters is written with empty parentheses: `() -> 4`.
- If the body consists of several statements, braces and `return` are needed: `(a, b) -> { int s = a + b; return s; }`.

The program below tries all of these forms with the same sum. Each lambda is assigned to a variable of an interface type. The type of a lambda is always such an interface.

```
import java.util.function.IntBinaryOperator;
public class Main {
    static int add(int x, int y) {
        return x + y;
    }
    public static void main(String[] args) {
        IntBinaryOperator t1 = (int x, int y) -> x + y;
        IntBinaryOperator t2 = (x, y) -> x + y;
        IntBinaryOperator t3 = (a, b) -> { int s = a + b; return s; };
        java.util.function.IntUnaryOperator twice = s -> s * 2;
        System.out.println(add(5, 3) + " " + t1.applyAsInt(5, 3) + " " + t2.applyAsInt(5,
3) + " "
                + t3.applyAsInt(5, 3) + " " + twice.applyAsInt(21));
    }
}
```

```
8 8 8 8 42
```

All four sums give 8; the one-parameter lambda `twice` doubles 21 to 42. The behavior is the same even though the notation changes.

4.5.4.2 The functional interface

In Java every value has a type. What is the type of a lambda expression? Java answers this question without adding a new kind of type: a lambda is an object of an interface that has a single abstract method.

Definition 4.5.7

A **functional interface** is an interface with a single abstract method. A lambda expression can be assigned only to such a type. The lambda writes the body of that one method; the parameter and return types are inferred from the signature of the method.

```
@FunctionalInterface
interface Greeting {
    void greet(String name);
}

Greeting s = name -> System.out.println("Hello " + name);
s.greet("Ayşe");
```

```
Hello Ayşe
```

The only abstract method of the `Greeting` interface is `greet(String name)`. The lambda `name -> System.out.println("Hello " + name)` is the body of this method. The compiler infers the type of `name` as `String` from the method's parameter. The return type is `void`; the `println` call produces no value either. Figure 4.5.11 shows the match.

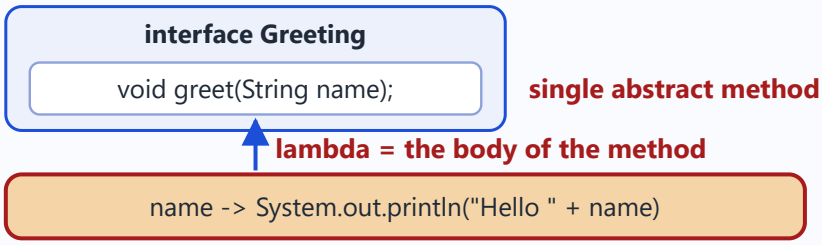


Figure 4.5.11: The single abstract method of the functional interface is filled in with the lambda expression. The lambda is the body of the method.

The `@FunctionalInterface` annotation is optional. If written, the compiler checks that the interface really is functional and does not allow a second abstract method to be added. Even without the annotation, every interface with a single abstract method can be a target for a lambda. The annotation is a safeguard against the programmer who changes the interface later. If someone accidentally adds a second method, all lambdas that use the interface break. The annotation catches this mistake in the interface itself.

The most important rule about lambdas is when they run. A lambda does **not run** when it is defined. The definition only creates an interface object and assigns it to the variable. The body runs when the method of the interface is called, that is on the line `s.greet(...)`. To see this, let us add numbered messages to the program:

```
@FunctionalInterface
interface Greeting {
    void greet(String name);
}
public class Main {
    public static void main(String[] args) {
        System.out.println("1: defining the lambda");
        Greeting s = name -> System.out.println("3: Hello " + name);
        System.out.println("2: not called yet");
        s.greet("Ayşe");
        s.greet("Ali");
    }
}
```

```
1: defining the lambda
2: not called yet
3: Hello Ayşe
3: Hello Ali
```

The order in the output shows the rule clearly. The lambda was defined before message 2, but its body runs after message 2. Since the same lambda is called twice, its body runs twice too. A lambda is a behavior that is stored like a value, carried around and called when wanted.

Assigning a lambda to a type other than a functional interface is a compile error. `Object` is a class and has no single abstract method:

```
public class LambdaObject {
    public static void main(String[] args) {
        Object o = () -> System.out.println("?");
    }
}
```

```
LambdaObject.java:3: error: incompatible types: Object is not a functional interface
        Object o = () -> System.out.println("?");
                   ^
1 error
```

An interface with two abstract methods cannot be a target of a lambda either. The compiler cannot know which method the lambda is the body of:

```
TwoAbstractLambda.java:7: error: incompatible types: Operation2 is not a functional
interface
        Operation2 i = (a, b) -> a + b;
                   ^
    multiple non-overriding abstract methods found in interface Operation2
1 error
```

4.5.4.3 Ready-made functional interfaces

There is no need to write a new interface for every lambda. The commonly used forms are ready in the package `java.util.function`. Table 4.5.6 gives the four most used.

Interface	Input	Output	Method	Example use
`Consumer<T>`	one value	none	`accept`	printing, storing
`Supplier<T>`	none	one value	`get`	producing a value
`Function<T, R>`	one value	one value	`apply`	converting

Interface	Input	Output	Method	Example use
`Predicate<T>`	one value	`boolean`	`test`	filtering, conditions

Table 4.5.6: The four basic interfaces of the package `java.util.function`.

The `<T>` and `<T, R>` beside the interface names are type parameters. `Function<String, Integer>` means a function that takes a `String` and returns an `Integer`. The types inside the angle brackets are the types of the input and the output.

```
Consumer<String> print = m -> System.out.println(m);
print.accept("Consumer ran");
Supplier<Integer> dice = () -> 4;
System.out.println(dice.get());
Function<String, Integer> length = m -> m.length();
System.out.println(length.apply("Giresun"));
Predicate<Integer> isPositive = x -> x > 0;
System.out.println(isPositive.test(-7));
```

```
Consumer ran
4
7
false
```

The program uses each of the four interfaces once. `Consumer` prints the message it receives and returns nothing. `Supplier` has no parameter; it produces 4 at every `get()` call. `Function` converts the string “Giresun” to its length: 7. `Predicate` answers the question “is it positive?” for −7: `false`. The method of each interface has a different name (`accept` , `get` , `apply` , `test`); the lambda does not know that name, the calling code does.

There are separate interfaces for primitive types too. `IntBinaryOperator` takes two `int` values and returns an `int` ; its method is `applyAsInt` . `IntUnaryOperator` takes a single `int` . These interfaces remove the cost of boxing and unboxing.

4.5.4.4 When does a lambda run?

Let us trace, in a list filtering example, how a lambda is stored like a value and called later. The lambda `isEven` says whether a number is even; the loop filters a list with this lambda.

```
Predicate<Integer> isEven = x -> x % 2 == 0;
List<Integer> numbers = Arrays.asList(3, 8, 5, 12);
List<Integer> result = new ArrayList<>();
for (int s : numbers) {
    if (isEven.test(s)) {
        result.add(s);
    }
}
System.out.println(result);
```

```
[8, 12]
```

Step	Line	Variables	New output
1	1: `Predicate<Integer> isEven = x -> x % 2 == 0;`	result=-	—
2	2: `List<Integer> numbers = Arrays.asList(3, 8, 5, 12);`	result=-	—
3	3: `List<Integer> result = new ArrayList<>();`	result=[]	—
4	4: `for (int s : numbers) {`	s=3, result=[]	—
5	5: `if (isEven.test(s)) {`	s=3, test=false, result=[]	—
6	4: `for (int s : numbers) {`	s=8, result=[]	—
7	5: `if (isEven.test(s)) {`	s=8, test=true, result=[]	—
8	6: `result.add(s);`	s=8, test=true, result=[8]	—
9	4: `for (int s : numbers) {`	s=5, result=[8]	—
10	5: `if (isEven.test(s)) {`	s=5, test=false, result=[8]	—
11	4: `for (int s : numbers) {`	s=12, result=[8]	—
12	5: `if (isEven.test(s)) {`	s=12, test=true, result=[8]	—
13	6: `result.add(s);`	s=12, test=true, result=[8, 12]	—
14	9: `System.out.println(result);`	result=[8, 12]	`[8, 12]`

Table 4.5.7: Trace of filtering with `Predicate`. The `test` column is the value the lambda returns at that step.

In the first step of Table 4.5.7 line 1 runs. The lambda is stored as a `Predicate<Integer>` object; its body has not run yet. `isEven` is a reference that points to this behavior. In the second step the list is built with the elements 3, 8, 5, 12, and in the third step the result list starts empty.

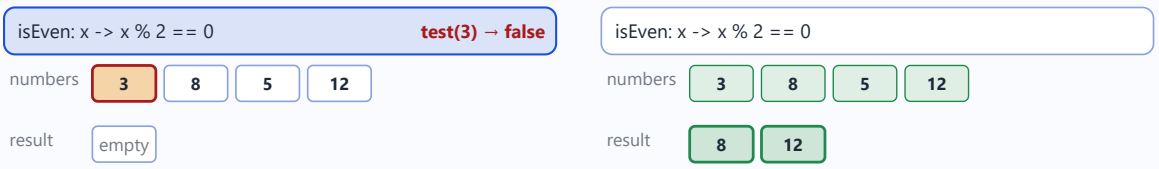


Figure 4.5.12: On the left the call `isEven.test(3)` runs the body of the lambda and returns `false`. On the right the loop has finished: the result list holds 8 and 12.

In the fourth step the loop takes the first element: `s = 3`. In the fifth step `isEven.test(3)` is called and the body of the lambda runs for the first time: `3 % 2 == 0` is false, the result is `false` (left of Figure 4.5.12). 3 is not added to the list. In the sixth and seventh steps the same work is done for 8: `8 % 2 == 0` is true. In the eighth step 8 is added to the result list. For 5 the condition is false, for 12 true; in the thirteenth step 12 is added too. In the last step the loop ends and `[8, 12]` is printed.

The lambda has run four times: once per `test` call. It never ran at the moment of definition. To verify this count, a counter can be added to the body of the lambda:

```
import java.util.*;
import java.util.function.*;
public class Main {
    static int counter = 0;
    public static void main(String[] args) {
        Predicate<Integer> isEven = x -> { counter++; return x % 2 == 0; };
        System.out.println("after definition: " + counter);
        List<Integer> numbers = Arrays.asList(3, 8, 5, 12);
        List<Integer> result = new ArrayList<>();
        for (int s : numbers) {
            if (isEven.test(s)) {
                result.add(s);
            }
        }
        System.out.println(result + " " + counter);
    }
}
```

```
after definition: 0
[8, 12] 4
```

Right after the definition the counter is 0; after the loop it is 4. Note that the counter is a `static` field, not a local variable. The reason is that local variables cannot be modified from inside a lambda.

4.5.4.5 Filtering and transforming with streams

The place where lambdas are used most is processing collections. The stream interface that came with Java 8 lets you pass a list through a series of operations, and each operation takes a lambda. The code below runs inside a `main` with `import java.util.*;`, `java.util.function.IntBinaryOperator` and `java.util.stream.Collectors` imported:

```
IntBinaryOperator add = (int x, int y) -> x + y;
System.out.println(add.applyAsInt(5, 3));

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
List<Integer> evens = numbers.stream().
                    filter(s -> s % 2 == 0).
                    collect(Collectors.toList());
System.out.println(evens);
System.out.println(numbers.stream().map(s -> s * s).toList());
```

```
8
[2, 4, 6]
[1, 4, 9, 16, 25, 36, 49]
```

The first two lines assign the sum lambda to the type `IntBinaryOperator` and call it with `applyAsInt(5, 3)`: 8. Then a list of the numbers 1 to 7 is built. `numbers.stream()` produces a stream from the list. `filter(s -> s % 2 == 0)` keeps only the elements for which the lambda returns `true`, that is the even numbers. `collect(Collectors.toList())` gathers the remaining elements in a new list: `[2, 4, 6]`. The lambda here has taken the place of an explicitly written loop; the Java library runs the loop inside `filter`.

The `map(s -> s * s)` on the last line transforms each element: every number is replaced by its square. The result is `[1, 4, 9, 16, 25, 36, 49]`. In short, `filter` reduces the number of elements but does not touch the values; `map` keeps the number of elements but changes the values. The `.toList()` on the last line came with Java 16. In older versions the same job is done with `collect(Collectors.toList())`. Here only the side of streams that takes a lambda is used.

4.5.4.6 Lambda and anonymous class

Before lambdas, a one-method behavior was written with an **anonymous class**. An anonymous class is a nameless inner class whose single object is created where it is defined. The program below implements the same `Runnable` interface in two ways:

```
Runnable r1 = new Runnable() {
    @Override
    public void run() {
        System.out.println("Anonymous class");
    }
};
Runnable r2 =
        () -> System.out.println("Lambda");
r1.run();
r2.run();
```

```
Anonymous class
Lambda
```

The anonymous class written for `r1` takes six lines: `new Runnable()`, braces, `@Override`, the full method signature and the body. The lambda written for `r2` does the same job in a single expression. The compiler interprets the lambda as the body of the single method `run()` of `Runnable`. The `run()` calls of the two objects print the expected messages.

The brevity of the lambda has a price; three limits must be known. First, a lambda can be assigned only to a functional interface type. An anonymous class can also implement an interface with several methods or extend a class.

Second, the local variables a lambda uses must be `final` or **effectively final**. Effectively final means a variable not written `final` but never assigned a value after its initial value. The lambda below reads the local variable `tax` and adds 20% VAT to a price of 150 TL: 150 + 150 · 20 / 100 = 180.

```
int tax = 20;
Function<Integer, Integer> withTax =
        price -> price + price * tax / 100;
System.out.println(withTax.apply(150)); // 180
tax = 18;  // if added: compile error
```

If the last line stays in a comment, the program compiles and prints 180. If that line is added, `tax` is no longer effectively final and the compiler points to the line where the lambda is:

```
LambdaFinal.java:6: error: local variables referenced from a lambda expression must be
final or effectively final
                price -> price + price * tax / 100;
                                         ^
1 error
```

The reason for the rule is that a lambda can run later. A lambda can be called long after the method where it was defined, even after that method has ended. So it carries not the local variable itself but a copy of its value. Had the variable changed later, the value the lambda sees and the value the method sees would diverge. Java forbids this ambiguity up front. Fields do not fall under this rule; a lambda can read and change a field. This is why the counter in the counter example was written as a `static` field.

Third, the keyword `this` means something different. The `this` inside a lambda refers to the enclosing object, that is the object of the method that wrote the lambda. The `this` in an anonymous class refers to the anonymous object itself:

```
public class Main {
    String name = "outer object";
    void demo() {
        Runnable lam = () -> System.out.println("lambda: " + this.name);
        Runnable anon = new Runnable() {
            String name = "anonymous object";
            public void run() { System.out.println("anonymous: " + this.name); }
        };
        lam.run();
        anon.run();
    }
    public static void main(String[] args) {
        new Main().demo();
    }
}
```

```
lambda: outer object
anonymous: anonymous object
```

Both bodies read `this.name`. Since the lambda has no object of its own, `this` is the outer `Main` object. The anonymous class is a separate object with its own `name` field.

4.5.4.7 Uses

Lambda expressions shorten code markedly in three kinds of work:

- **Collection operations.** Filtering, sorting, mapping a list and doing something for each element (`filter`, `sort`, `map`, `forEach`).
- **Thread management.** Giving briefly the body of a job to be run in parallel.
- **Readable code.** Instead of defining a separate method or class for a small operation used in one place, writing the operation where it is used.

```
List<String> names = new ArrayList<>(List.of("Zeynep", "Ali", "Mehmet"));
names.sort((a, b) -> a.length() - b.length());
System.out.println(names);
names.forEach(name -> System.out.println("Hello " + name));
Thread t = new Thread(() -> System.out.println("Thread ran"));
t.start();
```

```
[Ali, Zeynep, Mehmet]
Hello Ali
Hello Zeynep
Hello Mehmet
Thread ran
```

The call `names.sort(...)` takes the sorting criterion as a lambda. `(a, b) -> a.length() - b.length()` returns the difference in length of two names. If the difference is negative, `a` comes first; if positive, `b` comes first; if zero, the two count as equal. “Ali” goes first because it has three letters. “Zeynep” and “Mehmet” have six letters each; Java's list sort keeps the original order of equal elements, so “Zeynep” stays before “Mehmet”. Such a sort is called a stable sort.

`forEach` calls the given lambda for every element of the list and prints three greetings. The last two lines create a thread. The constructor of `Thread` expects a `Runnable`; `() -> ...` is a lambda with no parameters and becomes the body of `run()`. `t.start()` starts this body in a separate thread. The line `t.join()` was added to the program that was run; it makes `main` wait for the thread to finish. In this program the last message is printed last anyway, because there is no other work before it.

Changing the sorting criterion means only changing the lambda. The program below sorts the same list by three criteria. `Comparator.comparing(String::length)` is the ready-made way of saying “by length”; the notation `String::length` is a shorthand (a method reference) for the lambda `s -> s.length()`, which only calls one method.

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>(List.of("Zeynep", "Ali", "Mehmet"));
        names.sort(Comparator.comparing(String::length));
        System.out.println(names);
        names.sort((a, b) -> b.length() - a.length());
        System.out.println(names);

names.sort(Comparator.comparing(String::length).thenComparing(Comparator.naturalOrder()));
        System.out.println(names);
    }
}
```

```
[Ali, Zeynep, Mehmet]
[Zeynep, Mehmet, Ali]
[Ali, Mehmet, Zeynep]
```

The first line sorts from short to long, the second from long to short; in the second too, elements of equal length keep their original order. The third line looks at alphabetical order when lengths are equal and puts “Mehmet” before “Zeynep”.

4.5.5 Methods with bodies in interfaces

Until Java 8 an interface could contain only abstract methods and constants. This rigidity caused a big problem. New methods that work with lambdas (`forEach` , `removeIf` , `stream`) were to be added to Java's own `Collection` and `List` interfaces. Adding a new abstract method to an interface breaks all classes that implement it: every list class in the world would suddenly stop compiling. The solution was to allow methods with bodies in interfaces. Today an interface can contain three kinds of method with a body: default (`default`), `static` and `private` .

4.5.5.1 Default methods

Definition 4.5.8

A **default method** is a method written in an interface with the `default` keyword and a body (Java 8+). A class implementing the interface inherits it ready-made; it overrides it if it wishes.

The `Collection` interface below has three abstract methods and one default method. This `Collection` is a small interface we wrote ourselves; it should not be confused with `java.util.Collection` .

```
interface Collection {
    void add(Object o);
    void remove(Object o);
    int size();
    default boolean isEmpty() {
        return size() == 0;
    }
}
```

```
class Cart implements Collection {
    private int count = 0;
    public void add(Object o) { count++; }
    public void remove(Object o) { count--; }
    public int size() { return count; }
    // no isEmpty() written: the default version is used
}
// in main:
Cart cart = new Cart();
System.out.println(cart.isEmpty());
cart.add("apple");
cart.add("pear");
System.out.println(cart.size() + " " + cart.isEmpty());
```

```
true
2 false
```

`isEmpty()` calls the abstract `size()`. Whether a collection is empty is whether its element count is zero; this information is computed in the same way for every collection. Which `size()` runs is determined by the object. `Cart` has written only the three abstract methods; it takes `isEmpty()` ready-made. On an empty cart `true` is printed, and after two elements are added `2 false`.

The way a default method works is similar to a template method: the body in the interface calls the method of a class that has not been written yet. Let us trace a short version step by step.

```
interface Collection {
    void add(Object o);
    int size();
    default boolean isEmpty() {
        return size() == 0;
    }
}
class Cart implements Collection {
    private int count = 0;
    public void add(Object o) { count++; }
    public int size() { return count; }
}
Collection c = new Cart();      // main
System.out.println(c.isEmpty());
c.add("apple");
System.out.println(c.isEmpty());
```

```
true
false
```

Step	Line	Variables	New output
1	13: `Collection c = new Cart(); // main`	c=→Cart #1, count=0	—
2	14: `System.out.println(c.isEmpty());` 4: `default boolean isEmpty() {`	c=→Cart #1	—
3	5: `return size() == 0;` 11: `public int size() { return count; }`	size()=0	—
4	5: `return size() == 0;`	size()=0, result=true	`true`
5	15: `c.add("apple");` 10: `public void add(Object o) { count++; }`	count=1	—
6	16: `System.out.println(c.isEmpty());` 5: `return size() == 0;` 11: `public int size() { return count; }`	size()=1, result=false	`false`

Table 4.5.8: Trace of two calls of the default method `isEmpty()`.

In the first step of Table 4.5.8 a `Cart` object is created and assigned to `c`, which has the interface type. `Cart` has written only two abstract methods, `add` and `size`. In the second step `c.isEmpty()` is called. The JVM first looks for the method in the class of the object; `Cart` has no `isEmpty()`. The search goes up to the interface and the default body there runs.

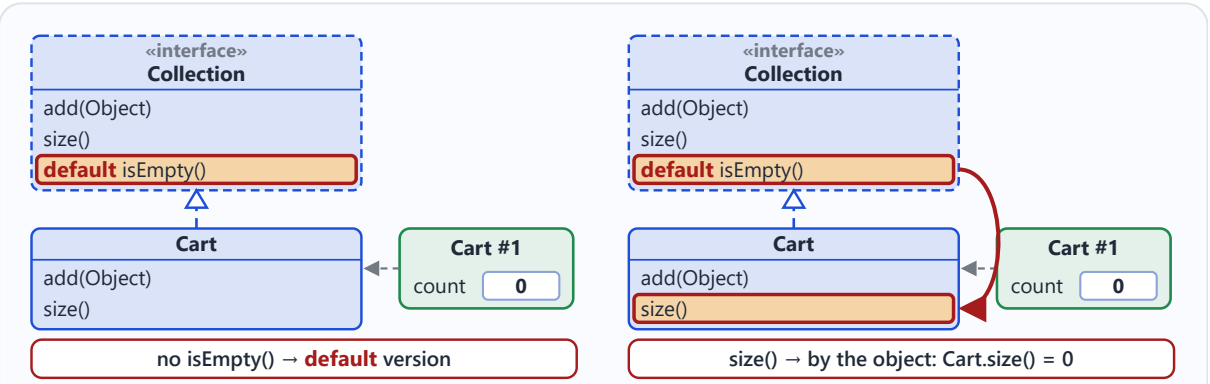


Figure 4.5.13: On the left, since `Cart` has no `isEmpty()`, the default version in the interface is chosen. On the right, the `size()` that the default body calls is bound dynamically to `Cart.size()`.

In the third step the default body calls `size()`. This call means `this.size()`, and `this` is a `Cart`; the object decides which body runs: `Cart.size()` returns 0 (Figure 4.5.13). In the fourth step the comparison `0 == 0` gives `true` and it is printed. In the fifth step `c.add("apple")` runs the version of `Cart` and `count` becomes 1. In the sixth step the same default body sees 1 this time and returns `false`.

The lesson of the example is this: a default method added after the interface was written reaches old classes too, without breaking them. Even if `isEmpty()` had not been known at all when `Cart` was written, `Cart` objects would start to use it the day it is added to the interface as a default method. Java's `forEach` and `removeIf` methods were added to all list classes this way.

There is one more precedence rule for default methods: **the class wins**. If a class receives a method with the same signature both from its superclass and from the default method of an interface, the version in the superclass is chosen:

```
class Machine {
    public void work() { System.out.println("Machine is working"); }
}
interface Printer {
    default void work() { System.out.println("Printing"); }
}
class Photocopy extends Machine implements Printer { }
public class Main {
    public static void main(String[] args) {
        Printer y = new Photocopy();
        y.work();
    }
}
```

```
Machine is working
```

`Photocopy` has not written `work()`. It could take the method both from `Machine` and from the default body of `Printer`. The rule puts the class hierarchy first and “Machine is working” is printed. A default method does not override behavior already present in the class; it only completes what is missing.

4.5.5.2 static methods

`static` methods can also be written in an interface (Java 8+). These methods are not tied to an object; they keep helper functions related to the interface together. They are called only with the name of the interface.

```
interface Geometry {
    static double circleArea(double r) { return Math.PI * r * r; }
    static double squareArea(double s) { return s * s; }
    static double perimeter(double s) { return s * 4.0; }
    static double boxVolume(double w, double h, double d) {
        return w * h * d;
    }
}
// in main:
System.out.println(Geometry.circleArea(2));
System.out.println(Geometry.squareArea(3));
System.out.println(Geometry.perimeter(2.5));
System.out.println(Geometry.boxVolume(2, 3, 4));
```

```
12.566370614359172
9.0
10.0
24.0
```

The `Geometry` interface gathers four geometric computations. The area of a circle with radius 2 is $\pi \cdot 2 \cdot 2 = 12.566370614359172$. The area of a square with side 3 is 9.0, and the perimeter of a square with side 2.5 is 10.0. The volume of a box of size $2 \times 3 \times 4$ is 24.0. The calls have the form `Geometry.circleArea(2)`.

The `static` methods of an interface are not inherited by the implementing class. This differs from `static` methods in classes; a `static` method of a class can also be called with the name of a subclass. Even if `Circle implements Geometry`, `Circle.squareArea(2)` does not compile:

```
interface Geometry {
    static double squareArea(double s) { return s * s; }
}
class Circle implements Geometry { }
public class StaticInheritance {
    public static void main(String[] args) {
        System.out.println(Circle.squareArea(2));
    }
}
```

```
StaticInheritance.java:7: error: cannot find symbol
        System.out.println(Circle.squareArea(2));
                                  ^
  symbol:   method squareArea(int)
  location: class Circle
1 error
```

The reason for the rule is multiple inheritance. A class can implement many interfaces; if two interfaces had a `static` method with the same name, which would `Circle.squareArea` refer to? By always requiring the interface name for the call, Java removes this question.

The constant π used in the area of the circle is in the class `Math` and must be written with its full name. If only `PI` is written, the compiler looks for it in the interface and cannot find it:

```
PIOnly.java:2: error: cannot find symbol
    static double circleArea(double r) { return PI * r * r; }
                                                ^
  symbol:   variable PI
  location: interface Geometry
1 error
```

4.5.5.3 Default conflicts

When a class implements two interfaces and both bring a default method with the same signature, the class cannot know which to take. In this case Java does not guess; it forces the class to write the method itself, that is, to override it. The version of the wanted interface can be called with the notation `InterfaceName.super.method()`.

```
interface Greeter {
    default void greet() { System.out.println("Hello"); }
}
interface Welcomer {
    default void greet() { System.out.println("Welcome"); }
}
class Clerk implements Greeter, Welcomer {
    public void greet() {       // must override
        Greeter.super.greet();
        Welcomer.super.greet();
    }
}
```

```
Hello
Welcome
```

`Clerk` wants to use both greetings. `Greeter.super.greet()` prints “Hello”, and `Welcomer.super.greet()` prints “Welcome”. The interface name before `super` says whose default body is wanted. Had `greet()` not been written at all in `Clerk`, the class would not compile:

```
DefaultConflict.java:7: error: types Greeter and Welcomer are incompatible;
class Clerk implements Greeter, Welcomer { }
^
  class Clerk inherits unrelated defaults for greet() from types Greeter and Welcomer
1 error
```

The office machine example below shows the same situation step by step. The machine is both a printer and a scanner, and both interfaces bring a default body for `work()`.

```
interface Printer {
    default void work() { System.out.println("Printing"); }
}
interface Scanner {
    default void work() { System.out.println("Scanning"); }
}
class OfficeMachine implements Printer, Scanner {
    public void work() {
        Scanner.super.work();
        Printer.super.work();
        System.out.println("Copy ready");
    }
}
Printer y = new OfficeMachine();   // main
y.work();
```

```
Scanning
Printing
Copy ready
```

Step	Line	Variables	New output
1	14: `Printer y = new OfficeMachine(); // main`	y=→OfficeMachine #1 (type: Printer)	—
2	15: `y.work();` 8: `public void work() {`	this=→OfficeMachine #1	—
3	9: `Scanner.super.work();` 5: `default void work() { System.out.println("Scanning"); }`	this=→OfficeMachine #1	`Scanning`
4	10: `Printer.super.work();` 2: `default void work() { System.out.println("Printing"); }`	this=→OfficeMachine #1	`Printing`
5	11: `System.out.println("Copy ready");`	this=→OfficeMachine #1	`Copy ready`
6	15: `y.work();`	y=→OfficeMachine #1	—

Table 4.5.9: Trace of `OfficeMachine.work()`, which combines two conflicting default methods.

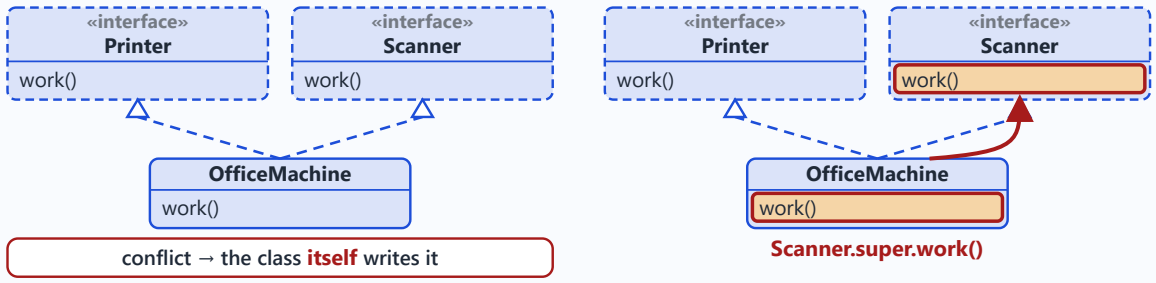


Figure 4.5.14: On the left two interfaces bring the same default method; the class itself has to resolve the conflict. On the right `Scanner.super.work()` calls the body of the named interface.

In the first step of Table 4.5.9 an `OfficeMachine` object is created and assigned to `y` of type `Printer`. In the second step `y.work()` is called. The type of the reference is `Printer` but the object is an `OfficeMachine`; the class's own `work()` runs. That the reference is a `Printer` does not mean the default body of `Printer` will be chosen; the version the class wrote always comes first.

In the third step `Scanner.super.work()` calls the default body of `Scanner` and "Scanning" is printed. In the fourth step it is the turn of `Printer`'s: "Printing". The class decides which versions are used in which order. In the fifth step the class makes its own addition: "Copy ready". The result is a new behavior that combines two abilities. In the sixth step the call ends.

Common mistake

Trying to call the default method of an interface with `super.method()`. Plain `super` refers to the superclass, not an interface; to reach the body of an interface, the name of the interface is written: `Scanner.super.work()`. This notation is valid only for interfaces the class implements directly.

4.5.5.4 private methods

As default methods multiply, shared code accumulates among them. This code needs to be moved to a separate method, but if the method is `default`, it is opened to every class that implements the interface. Java 9 solves this problem with `private` interface methods.

```
interface Printable {
    String content();
    default void print() {
        System.out.println(title() + ": " + content());
    }
    private String title() {   // Java 9+
        return "Document";
    }
}
class Report implements Printable {
    public String content() { return "Midterm results"; }
}
```

```
Document: Midterm results
```

`private` methods share the common helper code of default and `static` methods without opening it to the outside. They have bodies and can be called only from inside the interface. The flow is this: `print()` (default) first calls `title()` (`private`), then `content()` (the abstract method that `Report` implements) and combines the two (Figure 4.5.15).



Figure 4.5.15: The default method `print()` calls `title()`, which is visible only inside the interface, and the `content()` that `Report` wrote; the result is "Document: Midterm results".

The implementing class cannot see the `private` method. If `title()` is called from inside `Report`, the compiler looks for the method in `Report` and cannot find it:

```
PrivateCall.java:11: error: cannot find symbol
    public String content() { return title(); }
                                     ^
  symbol:   method title()
  location: class Report
1 error
```

4.5.5.5 Summary of interface members

All the members an interface can contain, their implicit modifiers and the Java version they came with are gathered in Table 4.5.10.

Member	Example	Implicit modifiers	Body	Java
Abstract method	`void draw();`	`public abstract`	none	1.0
Constant	`int SIDES = 4;`	`public static final`	—	1.0
Default method	`default boolean isEmpty() {…}`	`public`	yes	8
`static` method	`static double squareArea(…) {…}`	`public`	yes	8
Private method	`private String title() {…}`	only inside the interface	yes	9
Instance field, constructor	—	Not allowed: an interface has no objects and cannot hold state		

Table 4.5.10: Interface members.

The rule for the implementing class is short: all abstract methods of the interface are written `public`. Writing the default methods is optional; `static` and `private` methods are not inherited anyway. The ban in the last row of the table shows up in the compiler too. If a constructor is written inside an interface, the compiler does not recognize it as a constructor; it reads it as a method without a return type and expects a type name:

```
interface Counter {
    Counter() { }
}
public class InterfaceCtor { }
```

```
InterfaceCtor.java:2: error: <identifier> expected
    Counter() { }
         ^
1 error
```

4.5.6 Sealed types

When a class or interface is written, by default anyone can extend or implement it. This openness is often useful, but sometimes it does not reflect the truth of the design. A traffic light can only be red, yellow or green; a payment can only be by card, transfer or cash. In such a hierarchy, adding a new subtype unnoticed leaves all the code that handles it incomplete. Sealing, which came with Java 17, makes it possible to close the list of subtypes.

Definition 4.5.9

A **sealed class or interface** is a type declared with the `sealed` keyword (Java 17+). Only the subtypes named in its `permits` list can extend or implement this type.

Every permitted subtype must choose how the hierarchy continues after it. There are three options:

- `final` : the hierarchy ends here; no class can be derived from this type.
- `sealed` : the type is restricted again; it gives its own `permits` list.
- `non-sealed` : the restriction is lifted; anyone can extend this type.

The real benefit of sealing is that the compiler knows all the subtypes. Such a hierarchy is called a **closed hierarchy**. In the example below the `Animal` interface permits only `Cat` and `Dog` :

```
public sealed interface Animal permits Cat, Dog {
    void move();
}
final class Cat implements Animal {
    public void move() {
        System.out.println("Cat walks silently");
    }
}
non-sealed class Dog implements Animal {
    public void move() {
        System.out.println("Dog runs");
    }
}
class Kangal extends Dog {   // open
    public void move() {
        System.out.println("Kangal guards the flock");
    }
}
```

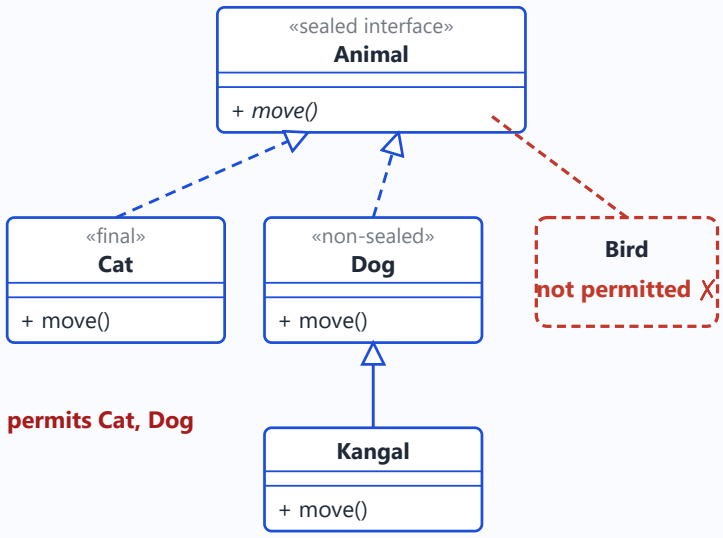


Figure 4.5.16: The sealed `Animal` interface permits only the types `Cat` (`final`) and `Dog` (`non-sealed`). `Kangal` extends `Dog` freely; `Bird` is not permitted.

Since `Cat` is `final`, that branch of the hierarchy is closed. `Dog` is `non-sealed`; so `Kangal` extends `Dog` without any permission (Figure 4.5.16). `Kangal` is indirectly an `Animal`, but not a direct subtype of `Animal`; so it need not be in the `permits` list.

What a sealed type gives the compiler shows when it is used with `switch`. In Java 21 a `switch` can branch on the type of an object, and the compiler checks whether all subtypes are handled:

```
// Java 21: switch covers all subtypes,
// no default branch needed
static String kind(Animal h) {
    return switch (h) {
        case Cat k -> "cat";
        case Dog k -> "dog";
    };
}
// in main:
Animal[] animals =
    { new Cat(), new Dog(), new Kangal() };
for (Animal h : animals) {
    h.move();
    System.out.println(kind(h));
}
```

```
Cat walks silently
cat
Dog runs
dog
Kangal guards the flock
dog
```

The `switch` in the method `kind` has no `default` branch. The compiler knows that an `Animal` can only be a `Cat` or a `Dog`; so it can prove that the two branches cover every case. The branch `case Cat k` catches the object under the name `k` if it is a `Cat`. The third object in the loop is a `Kangal`; `move()` runs its own version, while the `switch` recognizes it as a `Dog` and returns "dog".

The value of this check shows when the hierarchy changes. If a new subtype is added to the `permits` list, every `switch` that does not handle it stops compiling. The `switch` below handles only `Cat`:

```
sealed interface Animal permits Cat, Dog { }
final class Cat implements Animal { }
final class Dog implements Animal { }
public class SwitchMissing {
    static String kind(Animal h) {
        return switch (h) {
            case Cat k -> "cat";
        };
    }
}
```

```
SwitchMissing.java:6: error: the switch expression does not cover all possible input
values
        return switch (h) {
               ^
1 error
```

Had a `default` branch been written, this help would be lost: the new type would silently fall into `default`. Not writing `default` in a sealed hierarchy is therefore a deliberate choice.

Violating the sealing rules is a compile error. A class not in the permits list cannot implement the sealed type:

```
sealed interface Animal permits Cat, Dog { }
final class Cat implements Animal { }
final class Dog implements Animal { }
final class Bird implements Animal { }
public class SealedNotPermitted { }
```

```
SealedNotPermitted.java:4: error: class is not allowed to extend sealed class: Animal (as
it is not listed in its 'permits' clause)
final class Bird implements Animal { }
      ^
1 error
```

The phrase “sealed class” in the message is used for interfaces too. If a permitted subtype does not choose one of the three modifiers, the compiler stops too:

```
sealed interface Animal permits Cat, Dog { }
class Cat implements Animal { }
final class Dog implements Animal { }
public class SealedNoModifier { }
```

```
SealedNoModifier.java:2: error: sealed, non-sealed or final modifiers expected
class Cat implements Animal { }
^
1 error
```

The three fixes for this error are three different design decisions. `final class Cat` ends the hierarchy there. `sealed class Cat permits Tabby` restricts again; then `Tabby` must choose a modifier too. `non-sealed class Cat` opens `Cat` to everyone.

4.5.7 Interface or abstract class?

An abstract class and an interface are similar: no object can be created from either, both are used as reference types, and both make subtypes write methods. Since interfaces also carry methods with bodies after Java 8, the difference has become even finer. Table 4.5.11 puts the two tools side by side.

Property	Interface	Abstract class
Definition / use	`interface` · `implements`	`abstract class` · `extends`
Methods	Abstract; Java 8+ default and `static`, Java 9+ `private`	Both abstract and concrete methods
Fields	Only `public static final` constants; no instance fields	Any kind of field (instance fields included)
Constructor	None	May have one; the subclass calls it with `super(…)`
Access	Members implicitly `public` (except `private` helpers)	`public`, `protected`, `private`, package
Inheritance	A class implements many interfaces; an interface extends several interfaces	A class derives from only one class

Property	Interface	Abstract class
Purpose	A common ability for unrelated classes (“can do”)	Grouping closely related classes with common fields (“is a”)
Objects	No object can be created from either with `new`; both are used as reference types	

Table 4.5.11: Comparison of an interface and an abstract class.

The most decisive rows of the table are “Fields” and “Inheritance”. An abstract class carries state but has a single parent. An interface carries no state but a class can take as many interfaces as it wants. The choice is mostly made by these two criteria.

An **abstract class** is appropriate when:

- The subclasses are close relatives: `Square` and `Circle` are both a `Shape`.
- Common fields and a constructor are needed (like `name`).
- Part of the common code is ready-made (like a template method).

An **interface** is appropriate when:

- The same ability is to be given to unrelated classes: `Bird`, `Airplane` and `Drone` are all `Flyable`.
- Several abilities are to be given to one class.
- A contract is to be established between the using code and the implementing code. Such a contract is called an API (application programming interface).

Often the two are used together:

```
abstract class Animal implements Noisy { … }
class Duck extends Animal implements Flyable, Swimmable { … }
```

In this notation the abstract class `Animal` carries kinship and common state; the interfaces `Noisy`, `Flyable` and `Swimmable` add abilities. A `Duck` is an `Animal`, and can both fly and swim.

The reason Java does not allow multiple inheritance of classes is that, when two superclasses define the same field or method differently, it becomes unclear which one applies. This problem is called the **diamond problem** because of the shape of the drawing in which two superclasses come from a common ancestor. Since interfaces carry no state, most of the problem disappears. The only remaining ambiguity is the conflict of two default methods, and the compiler makes the class resolve that too.

Let us follow the decision process with an example. We are to design a `Vehicle` with a plate number and a constructor and an abstract method `int wheelCount()`, and a `Chargeable` with the constant `CAPACITY = 100` and the method `void charge(int percent)`. `Vehicle` has a field and a constructor; so it must be an abstract class. The ability to be charged can be given to an electric car, a phone and a drill alike; so `Chargeable` is an interface. The electric car uses both together:

```
abstract class Vehicle {
  private final String plate;
  Vehicle(String plate) { this.plate = plate; }
  abstract int wheelCount();
  String getPlate() { return plate; }
}
interface Chargeable {
  int CAPACITY = 100;
  void charge(int percent);
}
class ElectricCar extends Vehicle implements Chargeable {
  private int load = 20;
  ElectricCar(String plate) { super(plate); }
  int wheelCount() { return 4; }
  public void charge(int percent) {
    load = Math.min(CAPACITY, load + percent);
  }
  int getLoad() { return load; }
}
public class Main {
  public static void main(String[] args) {
    ElectricCar e = new ElectricCar("28 ABC 61");
    e.charge(50);
    System.out.println(e.getLoad());
    e.charge(50);
    System.out.println(e.getLoad());
    Vehicle t = e; Chargeable s = e;
    System.out.println(t.wheelCount() + " " + (s == t));
  }
}
```

```
70
100
4 true
```

The load starts at 20%. The first `charge(50)` makes it 70. In the second call 70 + 50 = 120 is computed, but `Math.min(CAPACITY, …)` limits the result to the capacity: 100. The constant `CAPACITY` comes from the interface by inheritance and is used by name inside the class. In the class declaration `extends` is written first, then `implements`, and `charge` is `public`. The object can be assigned to references of both type `Vehicle` and type `Chargeable`; since both references point to the same object, `s == t` gives `true`.

4.5.8 Generics

Interfaces and abstract classes let code work with different *classes*. Generics take the same idea one step further: they let a class or a method work with different *types*, without losing type safety. The angle brackets in `List<String>` and `Function<String, Integer>` are the syntax of generics.

4.5.8.1 Why generics?

Suppose we want to write a box class that can store a value of any type. Without generics, the only way that comes to mind is to use `Object`, the superclass of everything:

```
class ObjBox {
    private Object content;
    void put(Object o) { content = o; }
    Object get() { return content; }
}
ObjBox k = new ObjBox();
k.put("Hello");
String s = (String) k.get(); // cast required
k.put(42);                   // no complaint
String t = (String) k.get(); // runtime error
```

```
Exception in thread "main" java.lang.ClassCastException: class java.lang.Integer cannot be
cast to class java.lang.String (java.lang.Integer and java.lang.String are in module
java.base of loader 'bootstrap')
        at Main.main(Main.java:12)
```

This design has two flaws. First, reading a value from the box needs an explicit cast every time: `(String) k.get()`. The compiler does not know what is inside the box; it trusts the programmer's word. Second, when a value of the wrong type is put in, nobody objects. `k.put(42)` compiles, because 42 is boxed into an `Integer` object and every object is an `Object`. The error shows only while the program runs, on the line that tries to cast 42 to `String`: `ClassCastException`. The place where the mistake was made is far from the place where it is noticed.

```
class Box<T> {
    private T content;
    void put(T x) { content = x; }
    T get() { return content; }
}
Box<String> k = new Box<>();
k.put("Hello");
String s = k.get();          // no cast
k.put(42);                   // won't compile
```

```
GenericPut.java:10: error: incompatible types: int cannot be converted to String
        k.put(42);
              ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full
output
1 error
```

In the generic version the type of the box is a parameter: `Box<T>`. When `Box<String>` is written, `String` takes the place of `T`. `put` now accepts only `String` and `get` returns `String`. No cast is needed when reading. The wrong-typed `k.put(42)` is rejected at compile time, before the program runs.

The two gains of generics can be summed up this way. **Type safety**: a value of the wrong type is caught at compile time, not at run time. **Reuse**: a single implementation works for every type and no explicit cast is needed when reading.

4.5.8.2 Generic class and generic method

Definition 4.5.10

Generics means writing classes and methods with one or several **type parameters**. A type parameter (`T`) is a placeholder that is replaced by the **type argument** (`String` , `Integer` ...) given when the class or method is used.

```
class Box<T> {                    // T: type parameter
    private T content;
    void put(T x) { content = x; }
    T get() { return content; }
}
static <T> void print(T item) {    // generic method
    System.out.println("Item: " + item);
}
// in main:
Box<String> k1 = new Box<>();
k1.put("Hello");
Box<Integer> k2 = new Box<>();
k2.put(42);
int n = k2.get() + 1;               // unboxing
System.out.println(k1.get() + " " + n);
print(3.14);
print('A');
```

```
Hello 43
Item: 3.14
Item: A
```

`Box<String> k1` and `Box<Integer> k2` are two different uses of the same class. `k2.get()` returns an `Integer`; in the operation `+ 1` this value is unboxed and the result is 43. The empty pair of angle brackets in `new Box<>()` is called the **diamond**; the compiler infers the type from the declaration on the left.

`print` is a **generic method**. It declares the type parameter before the return type: `static <T> void print(T item)`. The type is not written at the call; the compiler infers it from the argument. In `print(3.14)` `T` is `Double`, in `print('A')` it is `Character`.

A type argument must be a reference type. `Box<int>` does not compile; for primitive types the wrapper classes are used (`Box<Integer>`):

```
GenericInt.java:6: error: unexpected type
        Box<int> k = new Box<>();
            ^
  required: reference
  found:    int
1 error
```

Type parameter names are traditionally a single capital letter: `T` type, `E` element, `K` key, `V` value, `R` result. The two letters in the name `Function<T, R>` describe the input type and the result type.

4.5.8.3 Bounded type parameters

About a plain `T` the compiler knows nothing; on a value of type `T` only the methods of `Object` can be called. If we want to write a generic method that computes twice a number, we have to say that `T` is a number.

Definition 4.5.11

A **bounded type parameter** is a type parameter written in the form `<T extends Parent>`. `T` can be only `Parent` or a subtype of it. `Parent` is called the **upper bound**. In return the method body can use the methods of `Parent`.

```
static <T extends Number> double twice(T x) {
    return x.doubleValue() * 2;   // a Number method
}
// in main:
System.out.println(twice(21));    // T = Integer
System.out.println(twice(1.5));   // T = Double
System.out.println(twice("ten")); // won't compile
```

```
42.0
3.0
```

Thanks to the bound `<T extends Number>`, the call `x.doubleValue()` compiles; this method is in the class `Number` and exists in all subclasses such as `Integer` and `Double`. For 21 `T = Integer` is inferred, for 1.5 `T = Double`, and the results are 42.0 and 3.0. On the third line `"ten"` is a `String` and not a subtype of `Number`; the compiler rejects the call:

```
TwiceTen.java:6: error: method twice in class TwiceTen cannot be applied to given types;
        System.out.println(twice("on"));
                           ^
  required: T
  found:    String
  reason: inference variable T has incompatible bounds
    upper bounds: Number
    lower bounds: String
  where T is a type-variable:
    T extends Number declared in method <T>twice(T)
1 error
```

The last part of the message shows the compiler's reasoning: for `T` the upper bound is `Number` and the lower bound coming from the argument is `String`. No type satisfies both.

An interface can be a bound too; even when the bound is an interface, `extends` is written, not `implements`. The method below finds the largest element in any comparable type (`Integer`, `String` ...). The bound `T extends Comparable<T>` guarantees that every value of type `T` can be compared with another `T` by the `compareTo` method.

```
static <T extends Comparable<T>> T max(T[] arr) {
    T best = arr[0];
    for (int i = 1; i < arr.length; i++) {
        if (arr[i].compareTo(best) > 0) {
            best = arr[i];
        }
    }
    return best;
}
// in main:
Integer[] numbers = {7, 12, 5};
String[] fruits = {"apple", "kiwi", "pear"};
System.out.println(max(numbers));
System.out.println(max(fruits));
```

```
12
kiwi
1 -1 6 -10
```

The `compareTo` contract is this: if `a.compareTo(b)` is negative, `a` is smaller; if zero, equal; if positive, larger. The third line of the output was added to the program to verify the four `compareTo` values used in the trace.

Step	Line	Variables	New output
1	11: `Integer[] numbers = {7, 12, 5};` 12: `String[] fruits = {"elderberry", "kiwi", "apricot"};`	numbers=[7, 12, 5], fruits=[elderberry, kiwi, apricot]	—
2	13: `System.out.println(max(numbers));` 1: `static <T extends Comparable<T>> T max(T[] arr) {` 2: `T best = arr[0];`	T=Integer, best=7	—
3	3: `for (int i = 1; i < arr.length; i++) {` 4: `if (arr[i].compareTo(best) > 0) {` 5: `best = arr[i];`	T=Integer, i=1, compareTo=1, best=12	—
4	3: `for (int i = 1; i < arr.length; i++) {` 4: `if (arr[i].compareTo(best) > 0) {`	T=Integer, i=2, compareTo=-1, best=12	—
5	8: `return best;` 13: `System.out.println(max(numbers));`	T=Integer, best=12	`12`
6	14: `System.out.println(max(fruits));` 2: `T best = arr[0];`	T=String, best="elderberry"	—
7	3: `for (int i = 1; i < arr.length; i++) {` 4: `if (arr[i].compareTo(best) > 0) {` 5: `best = arr[i];`	T=String, i=1, compareTo=6, best="kiwi"	—
8	3: `for (int i = 1; i < arr.length; i++) {` 4: `if (arr[i].compareTo(best) > 0) {`	T=String, i=2, compareTo=-10, best="kiwi"	—

Step	Line	Variables	New output
9	8: `return best;` 14: `System.out.println(max(fruits));`	T=String, best="kiwi"	`kiwi`

Table 4.5.12: Trace of the method `max` with two types.

In the first step of Table 4.5.12 two arrays are built. Both are suitable: `Integer` and `String` implement `Comparable` with their own types. The numbers in the array are boxed automatically. In the second step the compiler has inferred `T` as `Integer`. The first element is taken as the largest candidate: `best = 7`.

In the third step `12.compareTo(7)` is positive (1); 12 is larger and `best = 12`. In the fourth step `5.compareTo(12)` is negative (−1); `best` does not change. In the fifth step the loop ends and 12 is returned. The return type is `T`, that is `Integer`; no cast is needed.

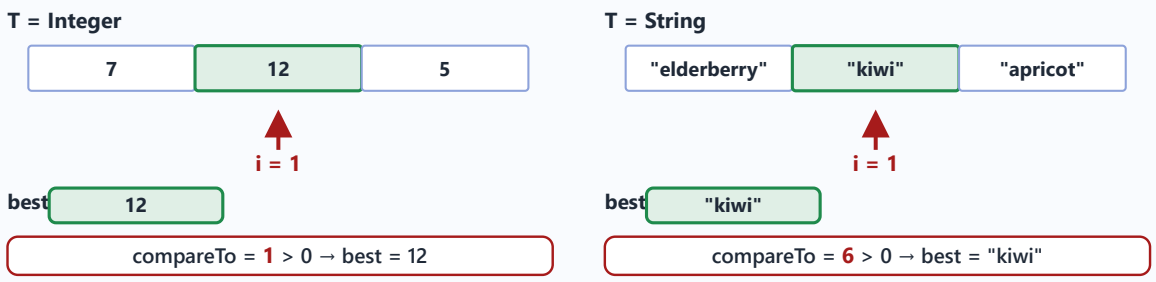


Figure 4.5.17: On the left `T = Integer`: since 12 is larger than 7, it becomes the new candidate. On the right `T = String`: since “kiwi” comes after “elderberry” in dictionary order, it becomes the new candidate.

In the sixth step the same method is called, this time with `T = String`. `compareTo` now compares strings in dictionary order: it returns the code difference of the first differing characters of the two strings. In the seventh step the first letters of “kiwi” and “elderberry” differ: 'k' − 'e' = 107 − 101 = 6. The result is positive and `best = "kiwi"` (Figure 4.5.17). In the eighth step “apricot” is compared with “kiwi”: 'a' − 'k' = 97 − 107 = −10; `best` does not change. In the last step “kiwi” is returned.

A single implementation has worked correctly with two different types. Thanks to the upper bound, the `compareTo` call is guaranteed at compile time: if `max` is called with an array of a type that has no `compareTo`, the program does not compile. It should not be forgotten that dictionary order rests on character codes. The codes of capital letters are smaller than those of lowercase letters; “Olive” comes before “apricot”.

4.5.8.4 Wildcards and invariance

`Integer` is a subtype of `Number`; an `Integer` can be used anywhere in place of a `Number`. Is `List<Integer>` then a subtype of `List<Number>` too? The first answer that comes to mind is “yes”; Java's answer is “no”:

```
import java.util.*;
public class Immutability {
    public static void main(String[] args) {
        List<Integer> ints = Arrays.asList(1, 2, 3);
        List<Number> x = ints;
    }
}
```

```
Immutability.java:5: error: incompatible types: List<Integer> cannot be converted to
List<Number>
        List<Number> x = ints;
                         ^
1 error
```

Definition 4.5.12

Invariance: even if `A` is a subtype of `B`, `List<A>` is not a subtype of `List<B>`. Generic types do not automatically inherit the subtype relationship between their type arguments.

The reason for the rule is writing. If a `List<Integer>` could be assigned to a `List<Number>` variable, 2.5 could be added to the list through that variable, because 2.5 is a `Number`. But the list is really a list of integers and a `Double` would have got into it. Code that reads from the list and expects `Integer` would crash at run time. Java closes this door at compile time (Figure 4.5.18).

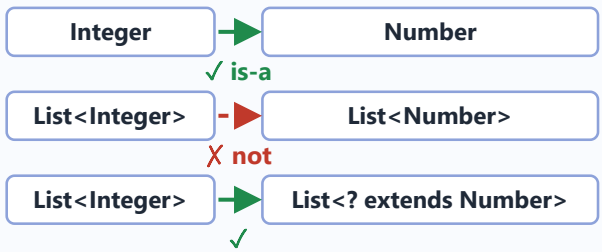


Figure 4.5.18: `Integer` is a subtype of `Number`; `List<Integer>` is not a subtype of `List<Number>`. The connection is made with a wildcard type: `List<Integer>` can be assigned to `List<? extends Number>`.

Invariance is safe but reduces flexibility. If a method that sums the values in a list of numbers takes a `List<Number>` parameter, a `List<Integer>` cannot be given to it. **Wildcards** solve this problem:

- `?`: any unknown type.
- `? extends T`: `T` or a subtype of it. Such a list *produces* values; it is read as `T`.
- `? super T`: `T` or a supertype of it. Such a list *consumes* values; `T` is written into it.

```
static double add(List<? extends Number> list) {
    double t = 0;
    for (Number n : list) t += n.doubleValue();  // read
    return t;
}
static void fill(List<? super Integer> list) {
    list.add(1);                                 // write
    list.add(2);
}
// in main:
List<Integer> ints = Arrays.asList(1, 2, 3);
List<Number> numbers = new ArrayList<>();
fill(numbers);
numbers.add(2.5);
System.out.println(add(ints) + " " + numbers);
// List<Number> x = ints;   won't compile
```

```
6.0 [1, 2, 2.5]
5.5
```

`add` only reads from the list; its parameter is `List<? extends Number>`. So a `List<Integer>`, a `List<Double>` and a `List<Number>` can all be given. Every element read is at least a `Number` and `doubleValue()` can be called. 1 + 2 + 3 = 6.0 is found. `fill` only writes to the list; its parameter is `List<? super Integer>`. Whatever list arrives can take an `Integer`. 1 and 2 are added to the `List<Number>` given to it; the program itself adds 2.5 and the list becomes `[1, 2, 2.5]`. The last line added to the program that was run shows that the same `add` works with this mixed list too: 1 + 2 + 2.5 = 5.5.

Trying to write to a list marked with `? extends` is a compile error. The compiler does not know the real type of the list; perhaps it is a `List<Double>` and 1 cannot be put into it:

```
import java.util.*;
public class ExtendsPrint {
    static void add(List<? extends Number> list) {
        list.add(1);
    }
}
```

```
ExtendsPrint.java:4: error: incompatible types: int cannot be converted to CAP#1
        list.add(1);
                 ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Number from capture of ? extends Number
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full
output
1 error
```

The `CAP#1` in the message is a name the compiler invents for the unknown type. There is a short rule for remembering which wildcard to choose: PECS (Producer Extends, Consumer Super). A structure that produces values (is read) is written with `extends`, one that consumes values (is written) with `super`.

4.5.8.5 Type erasure

Generic type information is mostly for the compiler. The compiler checks the type arguments, inserts the needed casts, and then erases this information. In the generated code `Object` (or the bound type in a bounded parameter) remains in place of `T`.

Definition 4.5.13

Type erasure is removing generic type arguments from the generated bytecode after they have been checked at compile time. At run time `Box<String>` and `Box<Integer>` are the same class.



Figure 4.5.19: In the source code there are `Box<String>` and a read without a cast. javac checks and erases the types; in the bytecode `Box` and the `(String)` cast that the compiler added remain.

```
class Box<T> {
    private T content;
    // T make() { return new T(); }   won't compile
}
// in main:
Box<String> a = new Box<>();
Box<Integer> b = new Box<>();
System.out.println(a.getClass() == b.getClass());
System.out.println(a.getClass().getName());
```

```
true
Box
```

The `getClass()` results of the two boxes are the same object: `true`. The name of the class is just `Box`; the type argument is not part of the name. Figure 4.5.19 shows the conversion the compiler makes: the compiler adds the `(String)` cast, which the programmer did not write, to the bytecode. Since the type check has passed, this cast never fails.

One consequence of type erasure is that `T` is unknown at run time. That is why `new T()` cannot be written; the JVM cannot know which class's object to create:

```
NewT.java:3: error: unexpected type
    T fresh() { return new T(); }
                        ^
  required: class
  found:    type parameter T
  where T is a type-variable:
    T extends Object declared in class Box
1 error
```

For the same reason a generic array cannot be created with `new T[10]`. The message is different this time:

```
NewTArray.java:2: error: generic array creation
    T[] arr = new T[10];
              ^
1 error
```

4.5.9 Annotations

Adding to code information that only the compiler or a tool will read is sometimes useful: “this method overrides a method of the supertype”, “this class should no longer be used”, “this method is a test”. Java uses annotations for this kind of information.

Definition 4.5.14

An **annotation** is metadata added with `@` to a class, method, field or parameter. An annotation does not change behavior by itself. The compiler, tools or the running program (through reflection) read it and act accordingly.

Annotation	Who reads it?	What does it do?
`@Override`	compiler	Reports an error if there is no such method in the supertype
`@Deprecated`	compiler	Obsolete element; code that uses it gets a warning
`@SuppressWarnings("unchecked")`	compiler	Silences the given warning
`@FunctionalInterface`	compiler	Checks that the interface has only one abstract method
`@Test` (JUnit)	test tool	Finds and runs the test methods

Table 4.5.13: Commonly used annotations.

Four of the annotations in the table are read by the compiler. `@Override` has proved useful several times in these notes: a method whose signature is misspelled is caught, thanks to the annotation, together with the real cause of the error. The check of `@FunctionalInterface` looks like this:

```
@FunctionalInterface
interface Operation {
    int apply(int a, int b);
    int negate(int a);   // 2nd abstract method
}
```

```
FuncTwo.java:1: error: Unexpected @FunctionalInterface annotation
@FunctionalInterface
^
  Operation is not a functional interface
    multiple non-overriding abstract methods found in interface Operation
1 error
```

Without the annotation the interface would compile without trouble; the error would appear only when someone tries to assign a lambda to this interface, in another file. The annotation writes the intention into the code and has the compiler check it.

The programmer can define their own annotations too. An annotation is defined with `@interface`; its elements are written like methods and can take a default value with `default`. The `@Task` annotation below records the author and priority of a method. While running, the program examines its own class, finds the annotated methods and calls them:

```
@Retention(RetentionPolicy.RUNTIME)  // read at run time
@Target(ElementType.METHOD)          // methods only
@interface Task {
    String author();
    int priority() default 1;
}
class Jobs {
    @Task(author = "Ayşe", priority = 3)
    public void backup() { System.out.println("Backup done"); }
    public void clean() { System.out.println("Cleaned"); }
}
// in main (reflection, java.lang.reflect.Method):
Jobs jobs = new Jobs();
for (Method m : Jobs.class.getDeclaredMethods()) {
    Task g = m.getAnnotation(Task.class);
    if (g != null) {
        System.out.println(m.getName() + ", "
            + g.author() + ", " + g.priority());
        m.invoke(jobs);
    }
}
```

```
backup, Ayşe, 3
Backup done
```

The two annotations above the annotation give information about the annotation itself. `@Target(ElementType.METHOD)` says that the annotation can be written only on methods. `@Retention(RetentionPolicy.RUNTIME)` says that the annotation is kept until run time. The loop goes over all the methods of the class `Jobs`. `getAnnotation(Task.class)` returns `null` if the method has no `@Task`. Only `backup` is annotated; its name, author and priority are printed, then `m.invoke(jobs)` calls the method and “Backup done” is printed. Since `clean` is not annotated, it is never called.

`@Retention`	**Where does the annotation stay?**
`SOURCE`	Only in the source code (like `@Override`); discarded after the compiler has read it
`CLASS`	In the `.class` file; reflection cannot see it (the default)
`RUNTIME`	Read by reflection while running

Table 4.5.14: The `@Retention` values.

The difference in Table 4.5.14 is easy to try. If the same program is run with the `@Retention` value of the annotation set to `CLASS`, the annotation stays in the `.class` file but reflection cannot see it:

```
import java.lang.annotation.*;
import java.lang.reflect.Method;
@Retention(RetentionPolicy.CLASS)
@Target(ElementType.METHOD)
@interface Task {
    String author();
    int priority() default 1;
}
class Jobs {
    @Task(author = "Ayşe", priority = 3)
    public void backup() { System.out.println("Backup done"); }
}
public class Main {
    public static void main(String[] args) throws Exception {
        int got = 0;
        for (Method m : Jobs.class.getDeclaredMethods()) {
            if (m.getAnnotation(Task.class) != null) got++;
        }
        System.out.println("annotation found: " + got);
    }
}
```

```
annotation found: 0
```

Definition 4.5.15

Reflection is a program examining and using its own classes, methods, fields and annotations while running. It is done with classes such as `Method` and `Field` in the package `java.lang.reflect`.

Reflection is a powerful tool; testing tools and large application frameworks find annotated methods this way. It has a price too: calls such as `m.invoke(...)` do not pass through the compiler's checks, a wrong call is noticed only while running, and the code runs more slowly. That is why reflection is used not in ordinary application code but where it is really needed.

4.5.10 Design patterns

Interfaces, abstract classes and lambdas are language tools. How to put these tools together for a given problem is a separate kind of knowledge. Experienced programmers have met the same design problems again and again and have given names to the solutions that work. All five patterns below are

explained with small console programs; four of them rely directly on an interface.

4.5.10.1 What is a pattern?

Definition 4.5.16

A **design pattern** is a tried and reusable solution template for a frequently met design problem. A pattern is not ready-made code to be copied; it is a shared vocabulary that names the problem, the solution and the trade-offs of the solution.

The common aim of patterns is to separate responsibilities and reduce dependencies between classes. A part with few dependencies can be changed without touching the others and can be tested on its own. Patterns are traditionally divided into three groups (Figure 4.5.20). **Creational** patterns deal with how objects are created, **structural** patterns with how objects are combined. **Behavioral** patterns arrange the cooperation of objects.

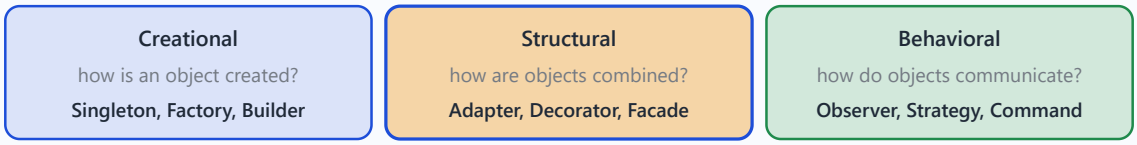


Figure 4.5.20: The three groups of design patterns and examples from each. Singleton and Factory are creational, Observer and Strategy behavioral; MVC is an architectural pattern.

A pattern is useful only if it solves a real problem. A pattern added needlessly complicates the program. Patterns are concrete applications of more general design principles such as “few dependencies between components” and “composition over inheritance”. **Composition** means a class, instead of inheriting behavior, holding another object as a field and having it do the work.

4.5.10.2 Singleton

Some classes should have a single object in the program. If the settings of the application, a log file or a hardware driver are kept in two copies, the copies may drift apart. The Singleton pattern ensures that the class has only one object and that this object is reached from a single place.

```
public final class Settings {
  private static final Settings INSTANCE = new Settings();
  private Settings() {
    System.out.println("Settings created");
  }
  public static Settings getInstance() { return INSTANCE; }
}
// main:
System.out.println("main started");
Settings a = Settings.getInstance();
Settings b = Settings.getInstance();
System.out.println(a == b);
```

```
main started
Settings created
true
```

The pattern has three parts. First, the constructor is `private`; `new Settings()` cannot be written outside the class:

```
SingletonNew.java:8: error: Settings() has private access in Settings
    Settings a = new Settings();
                 ^
1 error
```

Second, the single object is kept in a `private static final` field inside the class. Third, this object is reached through a `public static` method (`getInstance()`). Since both calls return the same object, `a == b` gives `true`.

The order of the output is instructive too: the message “main started” is printed before “Settings created”. The JVM initializes the `INSTANCE` field once, when the class `Settings` is first used, that is at the first `getInstance()` call. Since the JVM does class initialization only once, this method is safe with several threads too.

The Singleton pattern has a serious drawback: it behaves like a hidden global variable. Every class that calls `Settings.getInstance()` from inside becomes invisibly tied to that single object. When testing these classes it becomes hard to put another object in place of the real settings. Often a better way is to pass the object as a parameter to the constructor of the class that needs it. Giving a dependency from outside is called **dependency injection**.

4.5.10.3 Factory

Suppose a drawing program creates the shape the user chooses from a menu. The choice arrives as a string: “square”, “circle” … Deciding which class to create with an `if` chain everywhere scatters the creation code over the program. The Factory pattern gathers object creation in one place.

```
interface Shape { double area(); }
class Square implements Shape { … }   // area: a * a
class Circle implements Shape { … }  // area: π r²

class ShapeFactory {
  static Shape create(String kind, double size) {
    switch (kind.toLowerCase(Locale.ROOT)) {
      case "square":  return new Square(size);
      case "circle": return new Circle(size);
      default:      return null;  // unknown kind
    }
  }
}
Shape s = ShapeFactory.create("Square", 3);
System.out.println(s.area());   // 9.0
```

Definition 4.5.17

A **factory method** is a method that produces an object and returns it as a superclass or interface type. The calling code does not know which concrete class was created.

The method `ShapeFactory.create` takes the kind and the size, creates the suitable object and returns it as the interface type `Shape`. The calling code knows not `Square` but only `Shape`. The class bodies abbreviated with `…` in the code are completed in the program that was run:

```
import java.util.Locale;
interface Shape { double area(); }
class Square implements Shape {
  private final double a;
  Square(double a) { this.a = a; }
  public double area() { return a * a; }
}
class Circle implements Shape {
  private final double r;
  Circle(double r) { this.r = r; }
  public double area() { return Math.PI * r * r; }
}
class ShapeFactory {
  static Shape create(String kind, double size) {
    switch (kind.toLowerCase(Locale.ROOT)) {
      case "square":  return new Square(size);
      case "circle": return new Circle(size);
      default:      return null;  // unknown kind
    }
  }
}
public class Main {
  public static void main(String[] args) {
    Shape s = ShapeFactory.create("Square", 3);
    System.out.println(s.area());   // 9.0
    Shape u = ShapeFactory.create("triangle", 2);
    System.out.println(u == null);   // true
    System.out.println(ShapeFactory.create("CIRCLE", 1).area());
    System.out.println("CIRCLE".toLowerCase(Locale.forLanguageTag("tr-TR")) + " " +
"CIRCLE".toLowerCase(Locale.ROOT));
  }
}
```

```
9.0
true
3.141592653589793
cırcle circle
```

For “square” and 3 the area is 9.0. Adding a new shape changes only the factory and the new class; the code that asks for shapes stays the same. For an unknown kind the factory returns `null` and the calling code checks the result: for “triangle” `u == null` gives `true`.

The detail `toLowerCase(Locale.ROOT)` is important for Turkish users. If `toLowerCase()` is called without a parameter, it works according to the computer's locale. In the Turkish locale the lowercase of the capital “I” is the dotless “ı”; “CIRCLE” becomes “cırcle” and does not match the `"circle"`

branch. The last line of the program shows this difference: “cırcle” with Turkish rules, “circle” with the language-independent `Locale.ROOT`. Since the factory uses `Locale.ROOT`, the request “CIRCLE” produces the right shape and its area π · 1 · 1 = 3.141592653589793 is printed.

A factory is not needed everywhere. Writing a separate factory for objects that are already created simply is needless complexity; hiding an expression as clear as `new Point(3, 4)` behind a method only tires the reader.

4.5.10.4 Observer

A weather station measures the temperature. When a new measurement arrives, a display should show it, an alert module should warn when the temperature exceeds 30 degrees, and a logger should store the measurement. If the station calls these three parts directly, every new part changes the code of the station. The Observer pattern reverses this dependency.

Definition 4.5.18

The **observer pattern** establishes a one-to-many relationship between a **subject** object and the **observers** registered with it. When the state of the subject changes, all registered observers are notified. The subject does not know the concrete class of the observers; it only calls the common interface.

```
interface Observer { void update(int degrees); }

class Station {
  private final List<Observer> list = new ArrayList<>();
  void add(Observer g) { list.add(g); }
  void setTemp(int d) {
    for (Observer g : list) g.update(d);
  }
}
```

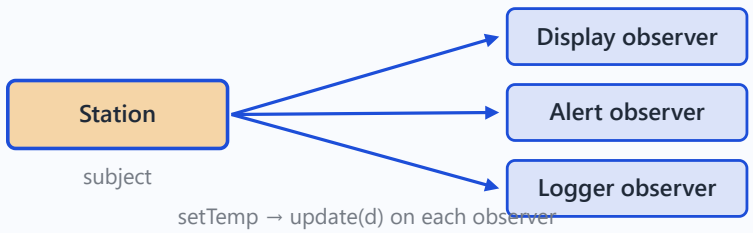


Figure 4.5.21: The station (the subject) calls the `update` method of every registered observer when the temperature changes; it does not know what the observers do.

`Observer` is an interface with a single method. `Station` keeps the observers in a list. `add` registers a new observer, and `setTemp` walks through the list and calls the `update` method of every observer. Since the interface has a single method, the observers can be written as lambdas too. In the program below two lambda observers are registered:

```
interface Observer { void update(int degrees); }
class Station {
  private final List<Observer> list = new ArrayList<>();
  void add(Observer g) { list.add(g); }
  void setTemp(int d) {
    for (Observer g : list) g.update(d);
  }
}
// main:
Station st = new Station();
st.add(d -> System.out.println("Display: " + d + " °C"));
st.add(d -> { if (d > 30) System.out.println("Alert: hot!"); });
st.setTemp(25);
st.setTemp(34);
```

```
Display: 25 °C
Display: 34 °C
Alert: hot!
```

Step	Line	Variables	New output
1	10: `Station st = new Station();`	st=Station, list=[]	—
2	11: `st.add(d -> System.out.println("Screen: " + d + " °C"));` 4: `void add(Observer g) { list.add(g); }`	list=[display]	—
3	12: `st.add(d -> { if (d > 30) System.out.println("Alert: hot!"); });` 4: `void add(Observer g) { list.add(g); }`	list=[display, alert]	—
4	13: `st.setTemp(25);` 6: `for (Observer g : list) g.update(d);`	d=25	—
5	6: `for (Observer g : list) g.update(d);` 11: `st.add(d -> System.out.println("Screen: " + d + " °C"));`	d=25, g=display	`Screen: 25 °C`
6	6: `for (Observer g : list) g.update(d);` 12: `st.add(d -> { if (d > 30) System.out.println("Alert: hot!"); });`	d=25, g=alert	—
7	14: `st.setTemp(34);` 6: `for (Observer g : list) g.update(d);`	d=34	—
8	6: `for (Observer g : list) g.update(d);` 11: `st.add(d -> System.out.println("Screen: " + d + " °C"));`	d=34, g=display	`Screen: 34 °C`

Step	Line	Variables	New output
9	6: `for (Observer g : list) g.update(d);` 12: `st.add(d -> { if (d > 30) System.out.println("Alert: hot!"); });`	d=34, g=alert	`Alert: hot!`

Table 4.5.15: Trace of the Observer pattern. “display” and “alert” are the names of the two registered lambda observers.

In the first step of Table 4.5.15 the subject object is created; the observer list is empty. In the second step the first observer is added. It is a lambda and implements the single method of the `Observer` interface. The lambda is added to the list but **does not run**. In the third step the second observer is added; this observer speaks only above 30 degrees.

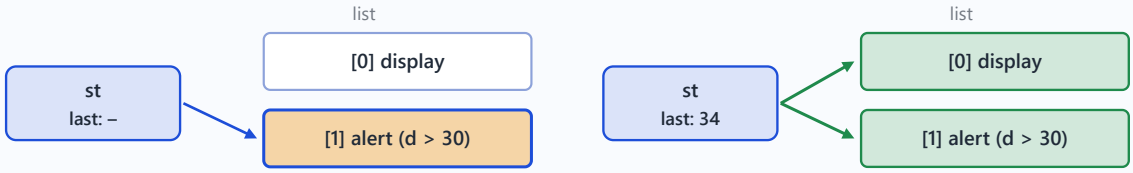


Figure 4.5.22: On the left two observers are registered in the list, but none has run yet. On the right `setTemp(34)` has reached the second observer in the list; since the condition is true, the alert is printed.

In the fourth step `setTemp(25)` is called and the loop walks through the list from the beginning. In the fifth step `update(25)` of the display observer is called and “Display: 25 °C” is printed. In the sixth step the alert observer is notified too, but since 25 > 30 is false, it prints nothing. In the seventh step a new measurement arrives: 34. In the eighth step the display prints the new value. In the ninth step the condition is true this time and “Alert: hot!” is printed (Figure 4.5.22).

The station has never known what any observer does. Adding a new observer does not change the code of the station. An observer does not have to be a lambda either; it can be an ordinary class that implements the interface. The logger below stores every measurement in a list:

```
import java.util.*;
interface Observer { void update(int degrees); }
class Station {
  private final List<Observer> list = new ArrayList<>();
  void add(Observer g) { list.add(g); }
  void setTemp(int d) {
    for (Observer g : list) g.update(d);
  }
}
class Logger implements Observer {
  private final List<Integer> measurements = new ArrayList<>();
  public void update(int degrees) { measurements.add(degrees); }
  List<Integer> getMeasurements() { return measurements; }
}
public class Main {
  public static void main(String[] args) {
    Station st = new Station();
    Logger k = new Logger();
    st.add(k);
    st.setTemp(25);
    st.setTemp(34);
    st.setTemp(18);
    System.out.println(k.getMeasurements());
  }
}
```

```
[25, 34, 18]
```

The logger has been registered with the station once and has received all three measurements. This example also shows that an observer can be tested independently of the subject: the logger's list is the proof that the station really passes on every measurement. The Observer pattern is used wherever “tell me when something happens” is needed, from button clicks in graphical interfaces to cell updates in spreadsheets.

4.5.10.5 Strategy

The checkout of a store computes the amount to pay. A 10% discount is applied to students, and in the campaign period 50 TL is deducted above 200 TL; tomorrow another rule may come. If the checkout knows these rules itself with an `if` chain, every new rule changes the code of the checkout.

Definition 4.5.19

The **strategy pattern** puts interchangeable algorithms behind a common interface. The object that uses the algorithm delegates the job to the strategy object given to it; when the strategy changes, its behavior changes without its code changing.

```
interface Discount { double apply(double amount); }

class StudentDiscount implements Discount {
  public double apply(double amount) { return amount * 0.90; }
}
class Checkout {
  private final Discount strategy;
  Checkout(Discount strategy) { this.strategy = strategy; }
  double due(double amount) {
    return strategy.apply(amount);
  }
}
Checkout k = new Checkout(new StudentDiscount());
System.out.println(k.due(250));   // 225.0
```

```
225.0
```

`Checkout` does not compute the discount itself. It keeps the `Discount` object given in the constructor in a field and delegates the job to it when `due` is called. `StudentDiscount` is an implementation of this interface: $250 \cdot 0.90 = 225.0$. Since the interface has a single method, the strategy can be a lambda too. The program below uses the same `Checkout` class with three different strategies:

```
interface Discount { double apply(double amount); }
class Checkout {
  private final Discount strategy;
  Checkout(Discount strategy) { this.strategy = strategy; }
  double due(double amount) { return strategy.apply(amount); }
}
// main:
Checkout student = new Checkout(t -> t * 0.90);
Checkout promo = new Checkout(t -> t >= 200 ? t - 50 : t);
System.out.println(student.due(250));
System.out.println(promo.due(250));
System.out.println(promo.due(120));
// fixed strategy: Checkout tested alone
System.out.println(new Checkout(t -> 7).due(100));
```

```
225.0
200.0
120.0
7.0
```

Step	Line	Variables	New output
1	8: `Checkout student = new Checkout(t -> t * 0.90);`	student=Checkout(10%)	—
2	9: `Checkout promo = new Checkout(t -> t >= 200 ? t - 50 : t);`	promo=Checkout(200 and above −50)	—
3	10: `System.out.println(student.due(250));` 5: `double due(double amount) { return strategy.apply(amount); }`	amount=250.0	`225.0`
4	11: `System.out.println(promo.due(250));` 5: `double due(double amount) { return strategy.apply(amount); }`	amount=250.0	`200.0`
5	12: `System.out.println(promo.due(120));` 5: `double due(double amount) { return strategy.apply(amount); }`	amount=120.0	`120.0`
6	14: `System.out.println(new Checkout(t -> 7).due(100));`	amount=100.0	`7.0`

Table 4.5.16: Trace of the Strategy pattern.

In the first step of Table 4.5.16 the 10% discount lambda is attached to the first checkout. In the second step the same `Checkout` class is built with another strategy: if the amount is 200 TL or above, 50 TL is deducted. In the third step `student.due(250)` delegates the job to the strategy: 250 × 0.90 = 225.0. In the fourth step the campaign strategy runs: since 250 ≥ 200, 250 − 50 = 200.0. In the fifth step 120 < 200 and there is no discount: 120.0. The behavior has changed without any change in the code of the checkout.

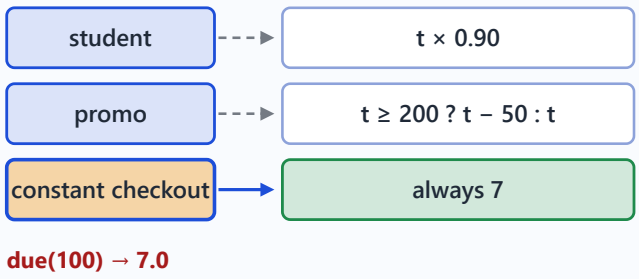


Figure 4.5.23: Three checkouts and their attached strategies: `t × 0.90`, `t ≥ 200 ? t − 50 : t` and a constant strategy that returns 7 whatever the amount.

The purpose of the sixth step is different. Instead of a real discount rule, a constant strategy that returns 7 whatever the amount is attached (Figure 4.5.23). The result is 7.0; the `int` value that the lambda `t -> 7` returns is widened because the return type of the method is `double`. This line tests

not the discount computation but only `Checkout` itself: does `Checkout` delegate the job to the strategy correctly? When the constant strategy is attached, the result can come only from the strategy; if 7.0 is seen, the delegation is correct. Testing one part alone by putting a simple fake object in place of the part it depends on has become possible because the strategy is given from outside.

The best-known example of a strategy in the Java library is sorting. The lambda in the call `names.sort((a, b) -> a.length() - b.length())` is a `Comparator` strategy; the sorting algorithm stays the same, the comparison criterion changes.

4.5.10.6 MVC

In programs that interact with the user, three different jobs tend to get mixed up: the data itself and its rules, showing the data on the screen, and reacting to the user's input. Putting these three jobs in one class means a small change made on the screen can break the data rules.

Definition 4.5.20

MVC (Model–View–Controller) is an architectural pattern that divides the program into three responsibilities. The **model** holds the data and the rules, the **view** only displays, the **controller** takes the input and manages the model and the view.

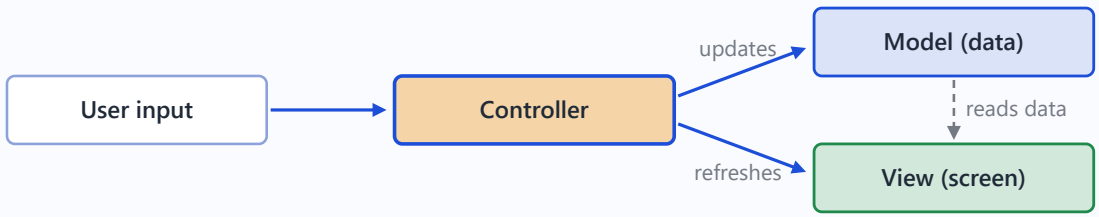


Figure 4.5.24: MVC: the controller takes the user's input, updates or reads the model and gives the result to the view. The model knows nothing of the screen.

Let us take the responsibilities one by one. The **model** (`Student` , `getName()`) contains the data and the rules. It knows nothing of the screen; it can be tested without a screen. The **view** (`StudentView`) only displays; its `show(name)` method writes to the console. The **controller** (`StudentController`) takes the input and manages both sides; `update()` reads the model and gives the result to the view.

```
Student m = new Student("Ayşe");
StudentView g = new StudentView();
StudentController d =
    new StudentController(m, g);
d.update();  // g.show(m.getName())
```

The full form of the three classes used in this short excerpt is in the program below:

```
class Student {
  private final String name;
  Student(String name) { this.name = name; }
  String getName() { return name; }
}
class StudentView {
  void show(String name) { System.out.println("Student: " + name); }
}
class StudentController {
  private final Student model;
  private final StudentView view;
  StudentController(Student model, StudentView view) {
    this.model = model;
    this.view = view;
  }
  void update() { view.show(model.getName()); }
}
public class Main {
  public static void main(String[] args) {
    Student m = new Student("Ayşe");
    StudentView g = new StudentView();
    StudentController d =
        new StudentController(m, g);
    d.update();  // g.show(m.getName())
  }
}
```

```
Student: Ayşe
```

The controller receives the model and view objects in its constructor. This too is dependency injection: the controller does not create the model and view it will work with; it receives them from outside. Changing the view, for example writing the name to a file instead of the console, affects only `StudentView`; the model does not change at all. Application frameworks implement MVC in different ways. There is no single "right" class structure; what they share is the separation of the three responsibilities.

4.5.10.7 When a pattern is not used

A side effect of learning patterns is the wish to use them where they are not needed. There are four common mistakes:

- **Making Singleton a global variable.** State that can be reached and changed from everywhere ties the parts of the program together invisibly.
- **A needless factory.** Writing a separate class for a creation as simple as `new Point(3, 4)`.

- **A needless observer.** Building an event system when a single direct method call is enough; the flow of the program becomes harder to follow.
- **Using a pattern for its name's sake.** The problem comes first, then the pattern. Before adding a pattern, two questions are asked: “Which change does it make easier? Which dependency does it reduce?”

Strategy and dependency injection make it possible to put a simple fake object in place of a real part. Attaching a constant strategy to the checkout and giving a fake payment object to the payment method are examples. Looked at the other way round, a class that is hard to test on its own often shows that its design should be reviewed.

4.5.11 Worked problems and exercises

4.5.11.1 Worked problems

The problems below ask for the rules of abstract classes and interfaces to be applied together. The solution to each problem gives the rule with its reason; the outputs and compiler messages have been verified by running them.

> **Example 4.5.1**
>
> For each of the fragments (a)–(e) below: does the fragment compile, and if not, what is the error? Of the fragments that compile, which one's object can be created with `new`?

```
// (a)
abstract class A {
    void f() { System.out.println("f"); }
}
// (b)
class B { abstract void g(); }
// (c)
interface I { int X = 1; }
...  I.X++;
// (d)
interface J { void h(); }
class C implements J { void h() { } }
// (e)
abstract class D { abstract void k(); }
abstract class E extends D { }
```

(a) compiles. `A` has no abstract method at all but is marked abstract; this is valid. **(b)** does not compile. `B` has an abstract method, and is not abstract itself:

```
P57b.java:1: error: B is not abstract and does not override abstract method g() in B
class B { abstract void g(); }
^
1 error
```

(c) does not compile. Since `X` is defined in an interface, it is implicitly `public static final`; `I.X++` tries to assign it a new value:

```
P57c.java:3: error: cannot assign a value to static final variable X
    void f() { I.X++; }
               ^
1 error
```

(d) does not compile. `h()` in `J` is implicitly `public`; `h()` in `C` has package access and narrows access:

```
P57d.java:2: error: h() in C cannot implement h() in J
class C implements J { void h() { } }
                            ^
  attempting to assign weaker access privileges; was public
1 error
```

(e) compiles. Since `E` is abstract too, it need not write `k()`; the obligation passes to the concrete subclasses of `E`. The answer to the second part of the question is "none". In the two fragments that compile, `A` and `E` are abstract; `new A()` and `new E()` are compile errors too:

```
P57new.java:8: error: A is abstract; cannot be instantiated
        A a = new A();
              ^
P57new.java:9: error: E is abstract; cannot be instantiated
        E e = new E();
              ^
2 errors
```

Example 4.5.2

What does the following program print? Why does `Fish` compile although it does not write `sound()`? What happens if `public` is removed from `sound()` in `Cat`? Does `Noisy s = new Fish(); s.name();` compile?

```
interface Noisy {
    default String sound() { return "..."; }
}
abstract class Animal implements Noisy {
    Animal() { System.out.print("A "); }
    abstract String name();
}
class Cat extends Animal {
    Cat() { System.out.print("C "); }
    String name() { return "cat"; }
    public String sound() { return "meow"; }
}
class Fish extends Animal {
    String name() { return "fish"; }
}
// in main:
Animal[] d = { new Cat(), new Fish() };
System.out.println();
for (Animal h : d)
    System.out.println(h.name() + ": " + h.sound());
```

```
A C A
cat: meow
fish: ...
```

When the array is built, two objects are created. In `new Cat()` the superclass constructor runs first and prints "A ", then the constructor of `Cat` prints "C ". In `new Fish()` there is no constructor written in `Fish`; the default constructor the compiler adds calls `super()` and only "A " is printed. So the first line becomes "A C A "; the `println()` ends the line. In the loop `name()` runs the version of each object's own class. `Cat` has overridden `sound()`: "meow". `Fish` has not; the default body of the interface runs: "...".

The reason `Fish` compiles is that `sound()` is a default method, not an abstract one. `Fish` inherits it. To be concrete it is enough for it to write `name()`, the only abstract method. If `public` is removed from `sound()` in `Cat`, the method tries to override the `public` version in the interface with narrower access:

```
P58public.java:9: error: sound() in Cat cannot implement sound() in Noisy
    String sound() { return "meow"; }
           ^
  attempting to assign weaker access privileges; was public
1 error
```

The last question tests the role of the reference type. The type of `s` is `Noisy`, and `Noisy` knows only `sound()`; even though the object has a `name()` method, the call does not compile:

```
P58swim.java:13: error: cannot find symbol
        s.name();
         ^
  symbol:   method name()
  location: variable s of type Noisy
1 error
```

Example 4.5.3

The constructor of an abstract class increases a `static` counter and the concrete method `summary()` calls the abstract `area()`. What does the program print?

```
abstract class Shape {
  static int count = 0;
  String name;
  Shape(String name) { this.name = name; count++; }
  abstract double area();
  String summary() { return name + ":" + area(); }
}
class Square extends Shape {
  double k;
  Square(double k) { super("Square"); this.k = k; }
  double area() { return k * k; }
}
class Circle extends Shape {
  double r;
  Circle(double r) { super("Circle"); this.r = r; }
  double area() { return 3 * r * r; }
}
public class Main {
  public static void main(String[] args) {
    Shape[] d = { new Square(2), new Circle(1) };
    for (Shape s : d) System.out.println(s.summary());
    System.out.println(Shape.count);
  }
}
```

```
Square:4.0
Circle:3.0
2
```

The element type of the array is the abstract `Shape`; the objects it points to are concrete. `new Square(2)` calls the constructor of `Shape` with `super("Square")`; no separate `Shape` object is created, the `Shape` part of the same object is built, and `count` becomes 1. `new Circle(1)` goes the

same way and `count` becomes 2. In the loop `summary()` comes from `Shape`; the `area()` call inside it is bound to the class of `this`. For `Square` 2 · 2 = 4.0, for `Circle` 3 · 1 · 1 = 3.0 is found. The last line prints the `static` counter that counted the two constructor calls: 2.

Example 4.5.4

Three unrelated classes implement the same `Priced` interface. At the end of the loop, what are the total and the price of the most expensive item?

```
interface Priced { double price(); }
class Book implements Priced {
  public double price() { return 40; }
}
class Pen implements Priced {
  int count;
  Pen(int count) { this.count = count; }
  public double price() { return 7.5 * count; }
}
class Ticket implements Priced {
  public double price() { return 25; }
}
public class Main {
  public static void main(String[] args) {
    Priced[] cart = { new Book(), new Pen(4),
                      new Ticket() };
    double total = 0;
    Priced mostExpensive = cart[0];
    for (Priced f : cart) {
      total += f.price();
      if (f.price() > mostExpensive.price()) mostExpensive = f;
    }
    System.out.println(total + " " + mostExpensive.price());
  }
}
```

```
95.0 40.0
```

The element type of the array is an interface. The only thing `Book`, `Pen` and `Ticket` have in common is the `Priced` contract. `mostExpensive` starts with the first element, the book. In the first round `f` points to the book: price 40.0, total 40.0; since 40 > 40 is false, the most expensive does not change. In the second round `f` points to the pen: 7.5 × 4 = 30.0, total 70.0; 30 > 40 is false. In the third round the ticket: 25.0, total 95.0; 25 > 40 is false. The output is “95.0 40.0”. The same `f.price()` call has run a method of a different class in each round. To add a new `Priced` class, the loop does not change.

Example 4.5.5

`incTwice()` is a default method written once in the interface. How many increments does it make on the two objects? What does the program print?

```
interface Counter {
  int value();
  void inc();
  default boolean isZero() { return value() == 0; }
  default void incTwice() { inc(); inc(); }
}
class Simple implements Counter {
  private int d;
  public int value() { return d; }
  public void inc() { d++; }
}
class Doubling extends Simple {
  @Override public void inc() {
    super.inc(); super.inc();
  }
}
public class Main {
  public static void main(String[] args) {
    Counter a = new Simple();
    Counter b = new Doubling();
    a.incTwice();
    b.incTwice();
    System.out.println(a.value() + " " + b.value()
        + " " + b.isZero());
  }
}
```

```
2 4 false
```

`a` is a `Simple`. In `a.incTwice()` the default body calls `inc()` twice; this call is `this.inc()` and is bound to the version of `Simple`: `d` becomes first 1, then 2. `b` is a `Doubling`. The same default body runs, but `this` is now a `Doubling`, and the overridden `inc()` calls `super.inc()` twice at each call: `d` becomes 2, then 4. The field `d` of `Doubling` comes from `Simple`; since it is `private`, it is changed only by the methods of `Simple`. Finally `isZero()` is a default method too; `value()` is 4 and the result is `false`. The output is “2 4 false”. A single default method has behaved differently on the two objects thanks to the methods the calling class overrides.

Example 4.5.6

The necessary imports have been made. What does the program print?

```
import java.util.*;
import java.util.function.*;
import java.util.stream.Collectors;
public class Main {
  public static void main(String[] args) {
    IntBinaryOperator f = (a, b) -> a * b + 1;
    IntUnaryOperator g = x -> x * x;
    int t = f.applyAsInt(3, 4) + g.applyAsInt(5);
    System.out.println(t);
    List<Integer> l =
            Arrays.asList(5, 8, 3, 12, 7);
    System.out.println(l.stream()
            .filter(s -> s > 5)
            .map(s -> s / 2)
            .collect(Collectors.toList()));
  }
}
```

```
38
[4, 6, 3]
```

`f.applyAsInt(3, 4)` runs the lambda with `a = 3`, `b = 4`: $3 \cdot 4 + 1 = 13$. `g.applyAsInt(5)` gives $5 \cdot 5 = 25$. Both values are `int` and their sum is 38. In the stream `filter(s -> s > 5)` keeps the elements greater than 5, preserving their order: 8, 12, 7. `map(s -> s / 2)` transforms each of them. The division is integer division: 8 / 2 = 4, 12 / 2 = 6, 7 / 2 = 3. The result is `[4, 6, 3]`.

4.5.11.2 Exercises

The exercises below require writing code. After each there is a possible solution and its result.

Exercise 1. An abstract `Employee` class: a `name` field, a constructor, an abstract `double salary()` and an `info()` that prints the name and salary. Subclasses: `FullTime`, which earns 30000 a month, and `Hourly`, which earns 120 hours × 200 TL. `info()` is called in an `Employee[]` array. Two small questions are answered in the same program: in the interface `Shape2D`, which has an abstract `double area()` and a default `summary()` returning `"Area: " + area()`, what does `summary()` return for `Triangle(6, 4)`? For `interface Account { static int add(int a, int b) … }` and `class Simple implements Account {}`, what do `Account.add(3, 4)` and `Simple.add(3, 4)` give?

```
abstract class Employee {
    private final String name;
    Employee(String name) { this.name = name; }
    abstract double salary();
    void info() { System.out.println(name + ": " + salary()); }
}
class FullTime extends Employee {
    FullTime(String name) { super(name); }
    double salary() { return 30000; }
}
class Hourly extends Employee {
    private final int hours;
    private final double fee;
    Hourly(String name, int hours, double fee) { super(name); this.hours = hours; this.fee
= fee; }
    double salary() { return hours * fee; }
}
interface Shape2D {
    double area();
    default String summary() { return "Area: " + area(); }
}
class Triangle implements Shape2D {
    private final double base, height;
    Triangle(double base, double height) { this.base = base; this.height = height; }
    public double area() { return base * height / 2; }
}
interface Account { static int add(int a, int b) { return a + b; } }
class Simple implements Account { }
public class Main {
    public static void main(String[] args) {
        Employee[] employees = { new FullTime("Ayşe"), new Hourly("Ali", 120, 200) };
        for (Employee c : employees) c.info();
        System.out.println(new Triangle(6, 4).summary());
        System.out.println(Account.add(3, 4));
    }
}
```

```
Ayşe: 30000.0
Ali: 24000.0
Area: 12.0
7
```

`info()` is concrete and written once in `Employee`; the `salary()` inside it runs the formula of each object's own class: 30000.0 and 120 · 200 = 24000.0. The area of the triangle is 6 · 4 / 2 = 12.0 and the default `summary()` returns "Area: 12.0". `Account.add(3, 4)` gives 7. `Simple.add(3, 4)` does not compile, because the `static` methods of an interface are not inherited:

```
SimpleAdd.java:5: error: cannot find symbol
        System.out.println(Simple.add(3, 4));
                           ^
  symbol:   method add(int,int)
  location: class Simple
1 error
```

Exercise 2. `sealed abstract class Shape permits Square, Circle` is written; `Square` is `final`, `Circle` is `non-sealed`. Do `Ellipse extends Circle` and `Triangle extends Shape` compile? In the same program a "fish that is both an animal and able to swim" is also designed.

```
sealed abstract class Shape permits Square, Circle { abstract double area(); }
final class Square extends Shape {
    double a = 2;
    double area() { return a * a; }
}
non-sealed class Circle extends Shape {
    double r = 1;
    double area() { return 3 * r * r; }
}
class Ellipse extends Circle { }
abstract class Animal { }
interface Swimmable { void swim(); }
class Fish extends Animal implements Swimmable {
    public void swim() { System.out.println("Fish is swimming"); }
}
public class Main {
    static String name(Shape s) {
        return switch (s) {
            case Square k -> "square";
            case Circle c -> "circle";
        };
    }
    public static void main(String[] args) {
        Shape[] arr = { new Square(), new Circle(), new Ellipse() };
        for (Shape s : arr) System.out.println(name(s) + " " + s.area());
        Swimmable y = new Fish();
        y.swim();
        Animal h = (Animal) y;
        System.out.println(h instanceof Swimmable);
    }
}
```

```
square 4.0
circle 3.0
circle 3.0
Fish is swimming
true
```

`Ellipse` compiles: since `Circle` is `non-sealed`, anyone can extend it. A `switch` recognizes an `Ellipse` as a `Circle` and says “circle”; its area comes from `Circle` too. `Triangle` is not in the permits list:

```
SealedClass.java:4: error: class is not allowed to extend sealed class: Shape (as it is
not listed in its 'permits' clause)
class Triangle extends Shape { }
^
1 error
```

Since `Square` is `final`, it is not possible to extend it either; this time the message speaks not of sealing but of `final`:

```
SealedFinalSub.java:4: error: cannot inherit from final Square
class BigSquare extends Square { }
                        ^
1 error
```

In the fish design there is both kinship and ability. `Animal` is an abstract class, and the ability to swim is the interface `Swimmable`: `class Fish extends Animal implements Swimmable`. The same object can be held as `Swimmable` and cast to `Animal`; both are different types of the same object.

Exercise 3. In `class Square implements GeometricShape`, `public void Draw() { … }` is written. What happens? What changes if `@Override` is added above the method?

`Draw` and `draw` are different names; Java distinguishes upper and lower case. `Draw()` is a new method and `draw()` is still missing: `Square is not abstract and does not override abstract method draw() in GeometricShape`. When `@Override` is added, the compiler gives a second message and points to the real place of the error:

```
NoOverride.java:4: error: Square is not abstract and does not override abstract method
draw() in GeometricShape
class Square implements GeometricShape {
^
NoOverride.java:5: error: method does not override or implement a method from a supertype
    @Override
    ^
2 errors
```

4.5.12 Common mistakes

The compiler catches almost all mistakes with abstract classes and interfaces. The messages are in English and formulaic; once recognized, the place of the error is found at once. The boxes below give the most frequent ones with their messages.

Common mistake

Creating an object from an abstract class. `new Shape("?")` → `Shape is abstract; cannot be instantiated`. The object is created from a concrete subclass; the type of the variable can stay abstract: `Shape s = new Square(3);`. The same message comes when an object is created from an interface.

Common mistake

Forgetting to write an abstract method. `Circle is not abstract and does not override abstract method draw() in Shape`. Either the method is written or the class is made `abstract`. When implementing an interface the message is the same; only the last word is the name of the interface.

Common mistake

Writing a body for an abstract method, or not making the class abstract. `abstract void draw() { }` → `abstract methods cannot have a body`. A method is either abstract or has a body. `class Shape { abstract void draw(); }` → `Shape is not abstract and does not override abstract method draw() in Shape`.

Common mistake

Contradictory modifiers. `abstract final`, `private abstract`, `static abstract` → `illegal combination of modifiers`. All three modifiers prevent overriding; an abstract method and an abstract class exist for subclasses.

Common mistake

Deriving from two classes. `class C extends A, B` → `'{' expected`. A class can have a single `extends` and many `implements`. That the message does not explain the error makes it harder to recognize.

Common mistake

Calling a method that is not in the type of the reference. `Flyable u = new Duck(); u.swim();` → `cannot find symbol`. Which methods can be called is decided not by the object but by the type of the reference. The method is called through a reference of a type that knows it (`Swimmable` or `Duck`).

Common mistake

Not resolving conflicting default methods. `class Clerk inherits unrelated defaults for greet() from types Greeter and Welcomer`. The method is overridden in the class; the version of the wanted interface can be called with `InterfaceName.super.greet()`.

Common mistake

Forgetting the sealing rules. A subtype that is not permitted: `class is not allowed to extend sealed class: Animal`. A permitted subtype with no modifier: `sealed, non-sealed or final modifiers expected`.

Common mistake

Changing a local variable that a lambda uses. `local variables referenced from a lambda expression must be final or effectively final`. A lambda carries a copy of the value of the local variable; the variable cannot be changed afterwards. A value that has to change is kept in a field.

Common mistake

Thinking generic types inherit the subtype relationship. `List<Number> x = ints;` (`ints` is a `List<Integer>`) → `incompatible types: List<Integer> cannot be converted to List<Number>`. A method that reads declares its parameter as `List<? extends Number>`, a method that writes as `List<? super Integer>`.

4.5.13 Summary

- **Abstraction** is hiding the detail and defining "what is done", leaving "how" to the subtypes. Java has two tools for it: the abstract class and the interface.
- An **abstract class** is defined with `abstract class`; no object can be created from it. It can contain fields, constructors, abstract and concrete methods. Its constructor builds the part of a subclass object that comes from it and is called with `super(...)`.
- An **abstract method** has no body and ends with `;`. A concrete subclass must implement it. It cannot be written together with `private`, `static` and `final`.
- The **template method** defines the skeleton of an algorithm in the superclass (in a `final` method); the varying steps are abstract methods written in the subclasses.
- An **interface** is a contract. Its methods are implicitly `public abstract`, its fields `public static final`; it has no instance fields and no constructor. A class can write a single `extends` and many `implements`; the implementing method must be `public`.
- The **type of the reference** decides which methods can be called, and the **class of the object** decides which body runs. Programming to an interface type gives loose coupling.
- A **lambda expression** is an anonymous function written in the form `(parameters) -> expression` and is assigned to a **functional interface** (an interface with a single abstract method) (Java 8+). It runs not when it is defined but when the interface's method is called. The local variables it uses must be effectively final.
- Methods with bodies in interfaces: **default** (Java 8, inherited), `static` (Java 8, called only with the interface name) and `private` (Java 9, only inside the interface). Conflicting default methods are overridden by the class; a method in the superclass beats a default method.
- A **sealed type** (`sealed`, Java 17+) limits its subtypes with `permits`; each subtype is `final`, `sealed` or `non-sealed`. In a closed hierarchy a `switch` can cover every case without a `default` branch.
- An abstract class groups closely related classes with common state and code ("is a"); an interface gives abilities to unrelated classes ("can do"). Often the two are used together.
- **Generics** write classes and methods with type parameters (`Box<T>`, `<T extends Parent>`). Type errors are caught at compile time. Generic types are invariant; flexibility comes from wildcards (`? extends`, `? super`). Type information is erased after compilation.
- **Annotations** are metadata like `@Override`; `RUNTIME` annotations are read by reflection.

- **Design patterns** are tried solution templates: Singleton (one object), Factory (gathering creation), Observer (notifying registered objects of changes), Strategy (an interchangeable algorithm) and MVC (separating data, display and control). A pattern is used only if it solves a real problem.

4.5.14 Key terms

abstraction abstract class abstract method concrete class template method constructor chaining interface implement multiple inheritance dynamic binding declared type / actual type loose coupling lambda expression functional interface anonymous class effectively final default method sealed class / interface generics bounded type parameter wildcard type erasure annotation reflection design pattern factory method dependency injection MVC (Model–View–Controller)

4.5.15 Test Yourself

1. The constructor of the abstract class `Document` prints `"D "`, and the constructors of the subclasses `Letter` and `Book` print `"L "` and `"B "` respectively. The method `final String label() { return name + "(" + pages() + ")"; }` of `Document` calls the abstract `pages()`; `Letter` returns 1, `Book(int chapters)` returns `chapters * 20`. What does the line `Document[] b = { new Letter(), new Book(3) };` print? Then what is the `label()` of each element?

   Answer

   In each object the constructor of the abstract superclass runs first, then that of the subclass: `D L D B`. No separate `Document` object is created. `label()` is written once in `Document` but `pages()` is bound to the class of each object: `letter(1)`, and with $3 \cdot 20 = 60$, `book(60)`.

2. The interface `Counter` has an abstract `int count()` and `default int twice() { return 2 * count(); }`; the interface `Meter` has `default int twice() { return 100; }`. The class `Both implements Counter, Meter` returns 7 in `count()` and writes `twice()` as `Counter.super.twice() + Meter.super.twice()`. What is `new Both().twice()`? What would happen if `Both` did not write `twice()`?

   Answer

   The default body of `Counter` calls `this.count()` and finds $2 \cdot 7 = 14$; that of `Meter` returns 100. The result is 114. If the method were not written, the class would not compile, because the two interfaces bring two default bodies with the same signature: `inherits unrelated defaults for twice()`.

3. After the definitions `Function<Integer, Integer> f = x -> x + 1;` and `Function<Integer, Integer> g = x -> x * 3;`, what are `f.apply(g.apply(4))` and `g.apply(f.apply(4))`? When the definition lines run, does either of the lambdas run?

   Answer

   The inner call runs first. `g.apply(4) = 12`, `f.apply(12) = 13`. `f.apply(4) = 5`, `g.apply(5) = 15`. The result is `13 15`. The definition lines only create two interface objects; the bodies run only when `apply` is called.

4. `l.removeIf(s -> s.length() == 4);` is applied to the list `List<String> l = new ArrayList<>(List.of("plum", "pear", "kiwi", "fig", "apricot"));`. What becomes of the list? Which functional interface is the parameter of `removeIf`?

   Answer

   The four-letter "plum", "pear" and "kiwi" are removed; the remaining list is `[fig, apricot]` and the order of the rest is preserved. `removeIf` takes a `Predicate`: the `test` method that returns `boolean` for each element. `removeIf` was added to the `Collection` interface as a default method in Java 8.

5. The generic method `max` of this chapter is called with `Integer[] d = {3, 9, 4}` and `String[] a = {"Olive", "apricot", "Apple"}`. What are the results? Why is the second result surprising at first sight?

   Answer

   For the numbers 9 is found. For the strings `compareTo` compares character codes, and the codes of capital letters (O = 79, A = 65) are smaller than those of lowercase letters (a = 97). So the string that comes last in dictionary order, the "largest", is `apricot`. The result is `9 apricot`.

6. The sealed interface `Animal` permits only `Cat` and `Dog`, and the `switch` in the method `kind` has two branches and no `default` branch. One day `Bird` is added to the `permits` list and `final class Bird implements Animal` is written. What happens? What would have happened if the `switch` had a `default` branch from the start?

   Answer

   The method `kind` stops compiling: `the switch expression does not cover all possible input values`. The compiler points to every place where the new type is not handled. With a `default` branch the program would compile and `Bird` would silently fall into `default`; the missing branch would be noticed only at run time, perhaps never.

7. In class `Machine`, `public void work()` prints "Machine is working". In the interface `Printer`, `default void work()` prints "Printing". For `class Photocopy extends Machine implements Printer { }`, what does `Printer y = new Photocopy(); y.work();` print? Why does the class compile?

   Answer

   "Machine is working" is printed. If a class receives a method with the same signature both from its superclass and from the default body of an interface, the method in the class hierarchy wins. In this case there is no conflict, so the class compiles. That the reference has type `Printer` does not change the result.

8. `Predicate<Integer> p = x -> { counter.add(x); return x > 2; };` is defined (`counter` is an empty `ArrayList<Integer>`). A loop over `List.of(1, 5, 2, 8)` increases `n` if `p.test(x)` is true. After the loop what are `n` and `counter`? What would happen if `counter` were an `int` local variable and the lambda increased it with `counter++`?

   Answer

   The lambda runs once per `test` call and adds the number it sees to the list. 5 and 8 satisfy the condition: `n = 2`, `counter = [1, 5, 2, 8]`. Adding an element to the list is not assigning a new value to the variable `counter`; the variable always points to the same list and stays effectively final. Increasing an `int` local variable with `counter++`, however, is assigning it a new value and does not compile: `local variables referenced from a lambda expression must be final or effectively final`.

4.5.16 Further reading

- Oracle, *The Java Tutorials*, the lessons "Interfaces and Inheritance", "Lambda Expressions" and "Generics": cover abstract classes, interfaces, default methods, lambdas and generics with short, runnable examples; online and free.
- J. Bloch, *Effective Java* (3rd ed., Addison-Wesley, 2018): Item 20 ("Prefer interfaces to abstract classes") discusses the choice between an interface and an abstract class, Item 31 the PECS rule for wildcards, and Item 42 the reasons for preferring lambdas to anonymous classes.
- E. Freeman and E. Robson, *Head First Design Patterns* (2nd ed., O'Reilly, 2020): explains the Strategy, Observer, Factory and Singleton patterns for beginners, with Java examples and plenty of drawings.
- E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994): the foundational book that established the vocabulary of design patterns; it defines the intent, structure and trade-offs of each pattern.

4.5.17 References

- Oracle, *The Java Tutorials*: “Abstract Methods and Classes”, “Interfaces”, “Default Methods”, “Lambda Expressions”, “Generics”, “Annotations”.
- J. Gosling et al., *The Java Language Specification*, Java SE 21 Edition (8.1.1.1 abstract Classes, 8.1.1.2 sealed, non-sealed, and final Classes, Chapter 9: Interfaces, 15.27 Lambda Expressions).
- Oracle, JEP 409: Sealed Classes (Java 17) and JEP 441: Pattern Matching for switch (Java 21).
- P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson (Chapter 10: Object-Oriented Programming: Polymorphism and Interfaces).
- K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly (Chapter 8: Interfaces and Abstract Classes).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson (Chapters 5–6).
- E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
- J. Bloch, *Effective Java*, 3rd ed., Addison-Wesley, 2018 (Items 20, 31 and 42).
- E. Freeman and E. Robson, *Head First Design Patterns*, 2nd ed., O'Reilly, 2020.

PART II · CHAPTER 5

Collections

Real programs work not with single variables but with masses of data. A class roster, the words of a dictionary, and the pending jobs of a printer all consist of many elements. The array is the first tool for this job, but its size is fixed in advance, and it leaves insertion, deletion, and searching to the programmer. Java offers a ready-made library for this need: the Collections Framework. This note covers the library as four families. Lists hold elements in order, queues organize elements waiting to be processed, sets prevent duplicates, and maps bind a key to a value. For each family we trace which internal structure it uses, how long each operation takes, and when to choose it. Next come the iterator, the ready-made algorithms of the `Collections` class, and the Stream API, which processes collections like a pipeline. All programs were compiled and run with JDK 21; the outputs, the values in the trace tables, and the compiler messages are taken from those runs.

LEARNING OUTCOMES

Getting to know the framework
Distinguish the interface, implementation, and algorithm elements of the Collections Framework; read the interface and class hierarchy; explain the benefit of declaring a variable with an interface type.

Working with generic types
Write type-safe code with generic types such as `List<String>`; use wrapper classes and autoboxing correctly for primitive types.

Choosing lists
Use the `ArrayList`, `LinkedList`, `Vector`, and `Stack` classes; draw their internal structures and compare the costs of their operations.

Using queues
Build first-in-first-out, priority, and last-in-first-out behavior with `Queue`, `PriorityQueue`, and `ArrayDeque`; trace how a binary heap works.

Choosing sets and maps
Choose among `HashSet`, `TreeSet`, `HashMap`, `TreeMap`, and their linked variants according to ordering and speed needs; write `equals` and `hashCode` for your own classes.

Traversing, sorting, streaming
Traverse and remove safely with an iterator; sort, shuffle, and search with `Collections`; write `Comparable` and `Comparator`; build filtering, mapping, and collecting pipelines with `java.util.stream`.

5.1 The Collections Framework

The array is the most basic structure for holding values of the same type one after another. It is fast and reaches any element instantly by index. But it has two major shortcomings. The first is its size: an array created with `new int[3]` holds three elements for its entire lifetime. The second is the lack of ready-made operations: to ask whether an element is in the array, to delete an element from the middle, or to sort the array, you must write a loop every time. This section introduces the library that closes both gaps and its basic ideas.

5.1.1 Why collections?

Definition 5.1

A **collection** is a data structure that stores and manages multiple elements in a single object. The **Java Collections Framework** is the whole of the ready-made interfaces, classes, and algorithms in the `java.util` package.

The first difference between a collection and an array is that its size is dynamic. The collection grows as elements are added and shrinks as they are removed. The second difference is the ready-made methods: `add` inserts, `remove` deletes, `contains` tells whether an element is present, and `size` gives the number of elements. These methods have been used and tested in millions of programs for years. A programmer who writes the same job by hand is far more likely to make mistakes than one who uses the library.

To see the difference concretely, let us try to store four numbers. If the array was created with three cells, the fourth value does not fit:

```
int[] arr = new int[3];      // size fixed: 3
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
// arr[3] = 40;  → ArrayIndexOutOfBoundsException
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds
for length 3
```

When the commented-out line `arr[3] = 40;` is executed, Java throws an `ArrayIndexOutOfBoundsException`. The message states the problem clearly: an array of length 3 has no index 3. The valid indices are 0, 1, and 2. The same four values go into a list without any trouble:

```
List<Integer> list = new ArrayList<>();
list.add(10);
list.add(20);
list.add(30);
list.add(40);       // no problem: it grows
System.out.println(list);
System.out.println("size: " + list.size());
```

```
[10, 20, 30, 40]
size: 4
```

Here `List<Integer>` is the type of the variable: "a list of integers". `new ArrayList<>()` creates the actual object. The list starts out empty, and each `add` call appends the value at the end. `println` prints the list in square brackets with the elements separated by commas; this format is produced by the list's own `toString` method. `size()` gives how many elements the list holds at that moment.

The analogy in the slides explains this difference well. An array is like a bus with a fixed number of seats: once the seats are full, no new passenger can board. An `ArrayList` is a bus that grows as passengers increase. How the growth happens, that is, how the bus expands internally, is traced in detail in the section on lists.

5.1.2 The four basic kinds

The dozens of classes in the framework fall into four basic kinds. Figure 5.1 shows each kind with a small example.

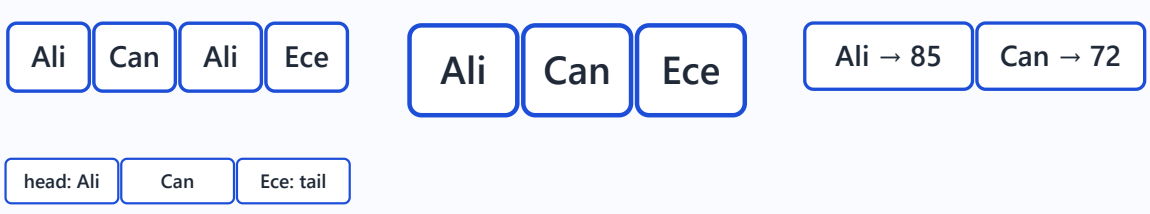


Figure 5.1: The four basic collection kinds. The list (Ali, Can, Ali, Ece) is ordered and allows duplicates. The set (Ali, Can, Ece) keeps only one of each element. The map binds keys to values (Ali → 85, Can → 72). In the queue, the element at the head leaves first.

- **List**: elements are kept in a definite order, and each has an index. The same element may appear more than once. Examples are `ArrayList` and `LinkedList`.
- **Set**: only unique elements are present. An element that already exists is not added a second time. Examples are `HashSet` and `TreeSet`.
- **Map**: key → value pairs are held. Each key appears once, while values may repeat. Examples are `HashMap` and `TreeMap`.
- **Queue**: elements waiting to be processed are held. Usually first in, first out (FIFO). Examples are `LinkedList` and `PriorityQueue`.

Besides these kinds, the framework pursues some common goals. The basic operations (insertion, removal, update, search) are performed with similar names in every kind, and all of them can be traversed with a for-each loop. The API is simple and consistent; whoever learns one kind can easily use the others. Each class is tuned for a particular data volume and access pattern. Thanks to generic types, an element of the wrong type is caught at compile time. Algorithms such as sorting, searching, and shuffling are ready-made, and misuse is reported with an exception. As a result, less code is written and the code that is written is more reliable.

5.1.3 The interface and class hierarchy

The skeleton of the framework consists of interfaces. An interface says which methods a type will have, but does not write the bodies of those methods. Figure 5.2 shows the basic interfaces and the extension relations among them.

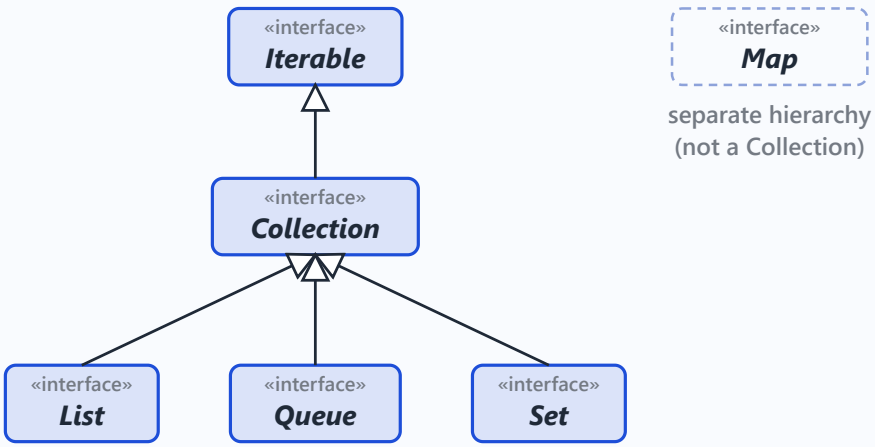


Figure 5.2: The hierarchy of the collection interfaces. `Collection` extends `Iterable`; `List`, `Queue`, and `Set` in turn extend `Collection`. `Map` is part of the framework, but does not extend `Collection`. The arrows with hollow triangle heads are UML notation and point to the extended super-interface.

At the top of the hierarchy stands `Iterable`. The only required method of this interface is `iterator()`. If a type is `Iterable`, it can be traversed with a for-each loop. `Collection` gathers the methods common to all collections: `add`, `remove`, `size`, `contains`, and others. `List`, `Queue`, and

`Set` inherit these common methods and add their own. For example, access by index (`get(i)`) exists only in `List`.

`Map`, on the other hand, is a separate branch. An element of a map is not a single value but a key-value pair. A method that takes a single element, such as `add(e)`, does not fit a map; in a map, insertion is done with `put(key, value)`. That is why `Map` does not extend `Collection` and is not `Iterable` either. This has an important consequence: a map cannot be given directly to a for-each loop. Instead, the keys, the values, or the pairs of the map are obtained as separate views and those are traversed.

Below the interfaces are the classes that implement them. Figure 5.3 gives the most commonly used classes and the interfaces they implement all together.

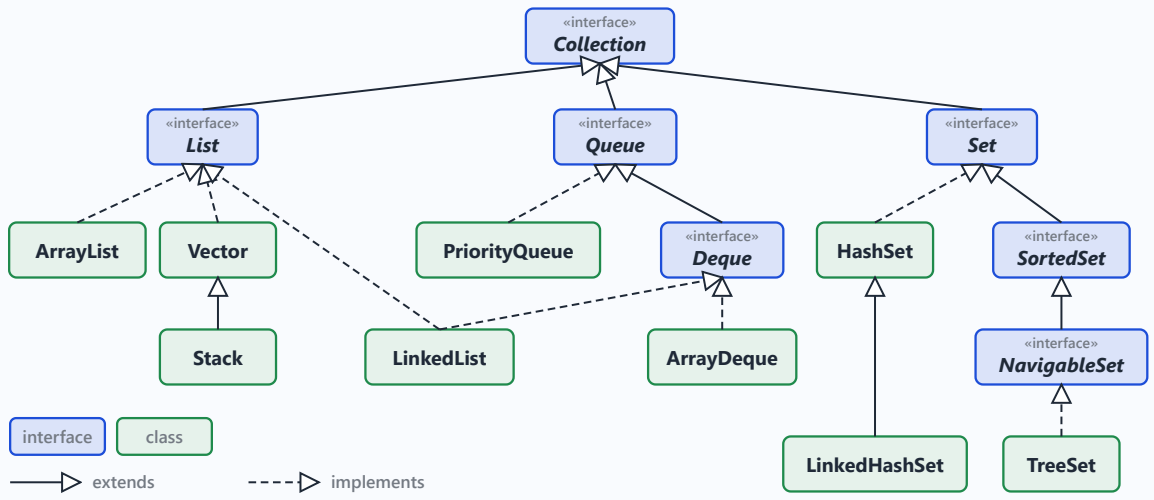


Figure 5.3: The hierarchy of the collection classes. Blue boxes are interfaces; they only define behavior. Green boxes are classes whose objects can be created with `new`. `LinkedList` implements both the `List` and the `Deque` interface.

The notable point in the figure is that a class can implement more than one interface. `LinkedList` is both a list (access by index) and a double-ended queue (insertion and removal at the front and the back). The same object offers the methods of whichever interface type the variable has. An object cannot be created from an interface type: if you write `new List<>()`, the compiler stops.

```
NewList.java:4: error: List is abstract; cannot be instantiated
        List<Integer> a = new List<>();
                          ^
1 error
```

The message says that `List` is abstract. An abstract type has no method bodies; the object is always supplied by a concrete class (`ArrayList`, `LinkedList`).

5.1.4 The three elements of the framework

The Collections Framework consists of three elements, each with a separate job.

Interface
: Defines how a data structure should behave, that is, it says *what* is to be done. `List` represents an ordered list: the elements are in a sequence, are accessed by index, and may repeat.

Implementation
: The concrete class that implements the interface, that is, it determines *how* the work is done. `ArrayList` implements the `List` interface with a dynamic array; `LinkedList` implements the same interface with linked nodes.

Algorithm
: Ready-made methods that do jobs such as sorting, searching, and shuffling. Most are found as `static` methods in the `Collections` class. `Collections.sort()` sorts a list.

The three elements can be seen together in a short piece of code:

```
List<String> list = new ArrayList<>();  // interface ← class
list.add("Red");
list.add("Blue");
Collections.sort(list);                 // algorithm
System.out.println(list);
```

```
[Blue, Red]
```

In the first line, the interface (`List<String>`) is on the left and the implementation (`ArrayList`) on the right. `Collections.sort` is the algorithm, and it arranges the list in its natural order, that is, alphabetical order for strings. Because "Blue" starts with B and "Red" with R, Blue comes first. The algorithm does not have to know which class the list belongs to; it works with the methods offered by the `List` interface. The same `sort` call also works on a `LinkedList` .

This arrangement is a modular design. Each component has a definite role and responsibility. Different implementations and algorithms can be used together through the same interface. Anyone who writes a new list class benefits from all the ready-made algorithms as soon as it implements the `List` interface.

5.1.5 Programming to an interface

An interface defines a behavior; classes implement this behavior with their own methods. Each class uses a different storage scheme: `ArrayList` a dynamic array, `LinkedList` a doubly linked list, `HashMap` a key-value table. Each meets the requirements of a particular data structure. A good habit for the programmer is to declare variables and parameters with the interface type.

```
public class PrintList {
    // interface parameter: works with any List
    static void print(List<String> list) {
        for (String s : list) {
            System.out.print(s + " ");
        }
        System.out.println("(" + list.size() + " items)");
    }

    public static void main(String[] args) {
        List<String> a = new ArrayList<>();
        a.add("Apple");
        a.add("Pear");
        List<String> b = new LinkedList<>(a);  // a copy
        b.add("Cherry");
        print(a);
        print(b);
    }
}
```

```
Apple Pear (2 items)
Apple Pear Cherry (3 items)
```

The parameter of the `print` method has type `List<String>`. The method does not know whether the list given to it is an `ArrayList` or a `LinkedList`, and it does not need to. The for-each loop and the `size()` call inside are available on every list. In `main`, `a` is an `ArrayList` and `b` is a `LinkedList`. The expression `new LinkedList<>(a)` creates a new list filled with the elements of `a`; so `b` first contains Apple and Pear, and then Cherry is added. Both calls go to the same method, and it prints objects of two different classes in the same way.

The real benefit of this approach shows when a change is needed. If one day it turns out that insertions at the front of the list are very frequent and `ArrayList` is too slow, only the `new` line is changed. Since the type of the variable is already `List`, the rest of the code stays the same. Had the variable been declared with type `ArrayList`, every method signature that uses that type would have to change as well. Joshua Bloch's book *Effective Java* summarizes this principle as "refer to objects by their interfaces".

Common mistake

Calling a class-specific method through an interface-typed variable. If the variable is declared as `List<String> l = new LinkedList<>();`, then `l.addFirst("x")` does not compile, because `addFirst` is defined in `Deque`, not in `List`. The compiler looks at the type of the variable, not at the actual class of the object. If you need to work at the ends, declare the variable with type `Deque<String>`.

5.1.6 Generic types and type safety

Definition 5.2

A **generic type** is a class or interface written with a type parameter. In `List<E>`, `E` stands for the element type; when you write `List<String>`, `String` takes the place of `E` and the type becomes "list of String". **Type safety** means that a value of the wrong type is caught at compile time.

```
List<String> names = new ArrayList<>();
names.add("Ali");
// names.add(42);           // compile error
String first = names.get(0);  // no cast needed

List<Integer> grades = new ArrayList<>();  // no List<int>
grades.add(85);               // boxing: int → Integer
int n = grades.get(0);        // unboxing → int
System.out.println(first + " " + n);
```

```
Ali 85
```

The element type is written inside the angle brackets (`<>`). `names` accepts only `String`. If the commented-out `names.add(42)` is uncommented, the program does not compile:

```
WrongType.java:5: error: incompatible types: int cannot be converted to String
        names.add(42);
                  ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full
output
1 error
```

The error is not postponed to run time; it is caught before the program ever runs. The second benefit is on the `get` side. The result of `names.get(0)` is directly a `String`; there is no need to write a cast.

Generics arrived with Java 5. In the earlier "raw" usage, the collection holds every element as an `Object`, and a wrong type shows up only at run time, as a `ClassCastException` when a read element is cast. A generic type catches the same mistake at compile time.

The empty `<>` on the right-hand side is called the **diamond** operator and has been available since Java 7. The compiler infers the type from the declaration on the left: `List<String> names = new ArrayList<>();` means the same as `new ArrayList<String>()`. The diamond is written only on the `new` side; `ArrayList<> a = …` does not compile.

Collections hold only objects

The type parameter must be a reference type; primitive types such as `int` and `double` cannot be written there. This is the message from an attempt at `List<int>`:

```
PrimitiveType.java:4: error: unexpected type
        List<int> grades = new ArrayList<>();
             ^
  required: reference
  found:    int
1 error
```

The message says "a reference type was expected, `int` was found". For primitive types, the corresponding **wrapper class** of each is used. Table 5.1 gives the most common ones.

Primitive type	Wrapper class
`int`	`Integer`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`
`long`	`Long`

Table 5.1: Primitive types and their wrapper classes.

The conversion between a primitive value and a wrapper object happens automatically. In the line `grades.add(85)`, `85` is an `int`, but the list expects an `Integer`. The compiler turns this call into `grades.add(Integer.valueOf(85))`; this is called **autoboxing**. In the line `int n = grades.get(0)`, a conversion in the opposite direction takes place: the value inside the `Integer` object is obtained with `intValue()`. This is called **unboxing**.

Unboxing has a trap. If the wrapper reference is `null`, there is no value to extract from it. The value of a key that is absent from a map comes back as `null`, and when this value is assigned to an `int`, the program crashes:

```
import java.util.*;
public class EmptyBox {
    public static void main(String[] args) {
        Map<String, Integer> m = new HashMap<>();
        int n = m.get("missing");
    }
}
```

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"java.lang.Integer.intValue()" because the return value of "java.util.Map.get(Object)" is
null
```

The message shows the source of the error exactly: the value returned by `Map.get` is `null`, so `intValue()` cannot be called. There is no visible method call on this line; the compiler itself inserted the `intValue()` call. That is why such errors surprise at first glance.

First examples: a list and a map

To see the two basic structures side by side, let us build a list of names and a phone book:

```
// list of names (ArrayList)
List<String> nameList = new ArrayList<>();
nameList.add("Ahmet");
nameList.add("Ayse");
System.out.println(nameList);

// phone book (HashMap)
Map<String, String> phoneBook = new HashMap<>();
phoneBook.put("Ahmet", "555-1234");
phoneBook.put("Ayse", "555-9876");
System.out.println(phoneBook.get("Ahmet"));
```

```
[Ahmet, Ayse]
555-1234
```

`ArrayList` keeps the elements in insertion order; `add` appends at the end each time. `HashMap`, on the other hand, maps a key (the name) to a value (the number). We insert into the map with `put(key, value)` and find the value with `get(key)`. `phoneBook.get("Ahmet")` does not scan the book from start to finish; it goes directly to a position computed from the key. How this speed is achieved is explained in the sections on sets and maps.

5.1.7 The methods of the Collection interface

The `Collection` interface defines the methods common to all lists, sets, and queues. Table 5.2 gives these methods, their purposes, and their return types.

Method	Purpose	Returns
`add(E e)`	adds the element to the collection	`boolean`: `true` if added
`remove(Object o)`	removes the element from the collection	`boolean`: `true` if removed
`size()`	gives the number of elements	`int`

Method	Purpose	Returns
`isEmpty()`	is the collection empty?	`boolean`
`contains(Object o)`	is the element in the collection?	`boolean`
`containsAll(Collection<?> c)`	are all elements of `c` in this collection?	`boolean`
`clear()`	removes all elements	—
`toArray()`	converts the collection to an array	`Object[]`
`iterator()`	gives an iterator for traversal	`Iterator<E>`
`forEach(Consumer<? super E> a)`	applies an action to each element (Java 8+)	—

Table 5.2: The basic methods of the `Collection<E>` interface. `E` is the element type; `?` (wildcard) means "any type".

That `add` returns a `boolean` may look pointless at first. Adding to a list always succeeds, and `add` always returns `true`. In a set, however, the situation is different: if the element is already present it is not added, and `add` returns `false`. The common interface is designed to cover both cases. `remove` and `contains` take their parameter as an `Object`; they test the equality of elements with the `equals` method.

In the following example, the type of the variable is deliberately chosen to be `Collection`:

```
Collection<String> colors = new ArrayList<>();
colors.add("red");
colors.add("green");
colors.add("blue");
System.out.println(colors.size());
System.out.println(colors.contains("green"));
colors.remove("green");
System.out.println(colors);
Collection<String> wanted = Arrays.asList("blue", "red");
System.out.println(colors.containsAll(wanted));
Object[] arr = colors.toArray();
System.out.println(arr.length);
colors.clear();
System.out.println(colors.isEmpty());
```

```
3
true
[red, blue]
true
2
true
```

The program proceeds line by line as follows. After three colors are added, `size()` is 3. `contains("green")` returns `true`. `remove("green")` finds the element by value and deletes it; the list becomes `[red, blue]`. `Arrays.asList("blue", "red")` creates a two-element list. `containsAll` asks whether every element of that list is in `colors`; order does not matter, and the result is `true`. `toArray()` copies the elements into a new array; the length of the array is 2. This array is independent of the collection: a change made in the array does not affect the collection. Finally, `clear()` deletes everything and `isEmpty()` returns `true`.

Because the variable has type `Collection`, only the common methods can be used. Even though the object is in reality an `ArrayList`, `colors.get(0)` does not compile:

```
CollGet.java:6: error: cannot find symbol
        String s = colors.get(0);
                          ^
  symbol:   method get(int)
  location: variable colors of type Collection<String>
1 error
```

The message says that a variable of type `Collection<String>` has no method named `get(int)`. This is the flip side of the rule of programming to an interface: the type of the variable determines which methods can be called. All the methods of `Collection`, on the other hand, exist on every `List`, `Set`, and `Queue` object.

5.2 Lists

The list is the most widely used collection. Most programmers learn `ArrayList` first after the array, and for most jobs they need nothing else. This section first gives the contract of the `List` interface. It then examines the two basic implementations, `ArrayList` and `LinkedList`, together with their internal structures. At the end of the section come `Vector` and `Stack`, left over from the first versions of Java, and the three ways of traversing a list.

5.2.1 The List interface

Definition 5.3

A **list** is a collection that keeps elements in insertion order, provides access to each element by an index, and allows the same element to appear more than once. The index of the first element is 0, and that of the last is `size() - 1`.

The word "order" in the definition can mean two different things, and the two are often confused. The elements of a list are **ordered**: each element has a definite position and stays at that position. But a list is not **sorted** by itself. Numbers added in the order 30, 10, 20 remain 30, 10, 20 in the list. To arrange them from smallest to largest, a sorting algorithm must be called separately.

The `List` interface adds index-based methods to the methods inherited from `Collection`. Table 5.3 gives the most commonly used ones.

Method	Purpose
`add(E e)`	appends at the end
`add(int i, E e)`	inserts at position `i`; later elements shift one place right
`get(int i)`	returns the element at `i`
`set(int i, E e)`	replaces the element at `i`, returns the old one
`remove(int i)`	removes and returns the element at `i`
`remove(Object o)`	removes the first element equal to `o`
`indexOf(Object o)`	the first position of `o`; −1 if absent
`size()`	number of elements

Table 5.3: The basic methods of the `List<E>` interface.

The interface has three well-known implementations. `ArrayList` uses a dynamic array and is very fast at access by index. `LinkedList` uses a doubly linked list; insertion and removal at the ends of the list are fast. `Vector` is an old, synchronized version; in new code `ArrayList` is preferred in its place.

Common mistake

Confusing `remove(int)` with `remove(Object)`. In a `List<Integer>`, both methods carry the same name. In the call `list.remove(1)`, the argument is an `int`, and the compiler picks the matching `remove(int index)` without boxing: the element at index 1 is deleted. To delete the element whose value is 1, the argument is passed as an object: `list.remove(Integer.valueOf(1))`. Examples are run in the section on common mistakes.

5.2.2 ArrayList and its internal array

Definition 5.4

A **dynamic array** is a structure that keeps its elements in an ordinary internal array and grows by copying into a larger array when that array becomes full. `ArrayList` implements the `List` interface with a dynamic array; it is derived from the `AbstractList` class.

```
ArrayList<String> fruits = new ArrayList<>();
fruits.add("Apple");
fruits.add("Pear");
System.out.println(fruits.get(1));  // by index
```

```
Pear
```

In this example the variable is deliberately declared with type `ArrayList`; the slides show both spellings. `fruits.get(1)` returns the element at index 1, that is, the second fruit added. Since indices start at 0, "Apple" is at 0 and "Pear" at 1.

The speed of `ArrayList` comes from its internal array. The `get(i)` call goes directly to cell number `i` of the array. Every cell in the array has the same size, and the cells lie one after another in memory. So the address of cell `i` is found with a single multiplication and addition. Whether the list has ten elements or ten million, this job takes the same time. `set(i, e)` is fast for the same reason.

Big O notation is used to describe costs. **O(1)** says that the time is independent of the number of elements, that is, constant. **O(n)** says that the time grows in proportion to the number of elements `n`: when the number of elements doubles, the time also roughly doubles. `get` and `set` are O(1). Appending at the end is O(1) on average. Insertion into the middle and removal from the middle are O(n) because they shift elements.

How does ArrayList grow?

The length of the internal array is called the **capacity**. Capacity is not the same thing as the `size()` of the list. `size()` is the number of filled cells; capacity is the total number of cells. As long as empty cells remain, the list appends directly. When the array is full, a larger array is allocated and the old elements are copied into it. Figure 5.4 shows this process on a list with capacity 4.

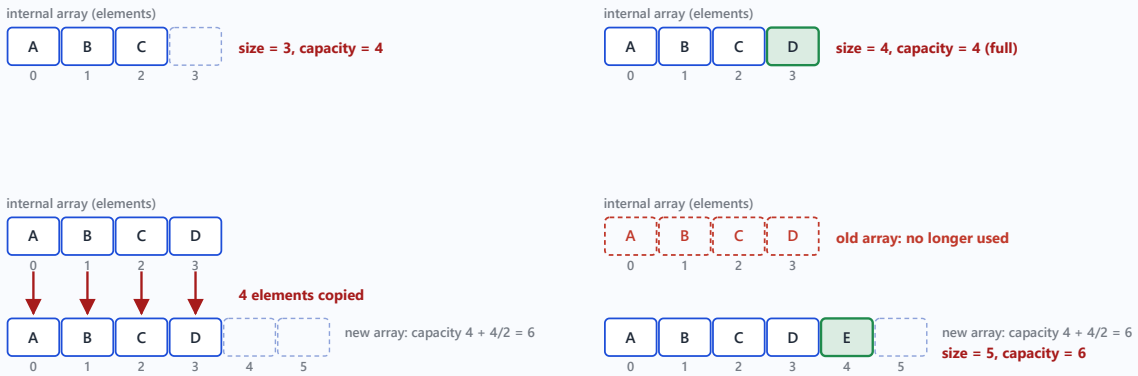


Figure 5.4: The growth of an `ArrayList`. At top left, the 4-cell internal array of a list created with `new ArrayList<>(4)` holds A, B, C. At top right, `add("D")` writes into the empty cell and no element is moved. At bottom left, when `add("E")` arrives the array is full: a new 6-cell array is allocated and the 4 old elements are

copied one by one. At bottom right, E is written into cell 4 of the new array; the old array is left to the garbage collector.

The length of the new array is about 1.5 times the old capacity: `4 + 4/2 = 6`. We can observe this rule ourselves. The internal array of `ArrayList` is in a private field called `elementData`. Normal programs do not access this field, but for measurement it can be read with reflection. The following output is the result of this note's measurement program on JDK 21:

```
size 3 cap 4
size 4 cap 4
size 5 cap 6
empty cap 0: 1->10 11->15 16->22 23->33 34->49 50->73 74->109
after clear cap 109
```

The first three lines are exactly the same as the steps in the figure: with capacity 4, three and four elements fit, and at the fifth element the capacity becomes 6. The fourth line shows the behavior of the no-argument `new ArrayList<>()`. An empty list allocates no cells at all at first (capacity 0). The first `add` allocates a 10-cell array; this is what "default capacity 10" in the slides means. Later growths go to 15, 22, 33, 49, 73, and 109 cells. Each is 1.5 times the previous one, with the fraction discarded. The last line shows an important detail: `clear()` deletes the elements but does not shrink the internal array. Deleting does not reduce the capacity.

Amortized O(1)

At the moment of growth, the `add` call copies all the elements; this single call takes O(n). Even so, appending at the end is said to be "O(1) on average". The reason is that copying is rare. Because the capacity grows by a factor of 1.5 each time, the number of additions between two growths also keeps increasing. Dividing the total number of copies by the `n` additions gives a constant number. This note's counting program computes it directly:

```
10 adds: 0 copies, ratio 0.00
100 adds: 202 copies, ratio 2.02
1000 adds: 2456 copies, ratio 2.46
1000000 adds: 2430972 copies, ratio 2.43
```

Over a million additions, about 2.4 million elements are copied. On average 2.43 copies fall on each addition, and this ratio does not increase as `n` grows. Whatever the number of additions, each addition does a constant amount of work on average. This kind of average is called **amortized** cost. A single call is sometimes expensive, but over a long sequence of calls the share of the expensive steps stays constant.

5.2.3 Tracing ArrayList methods

To see how the list methods work together, let us build a list with five names and make changes to it:

```
ArrayList<String> names = new ArrayList<>();
names.add("Ali");
names.add("Banu");
names.add("Cem");
names.add("Deniz");
names.add("Ece");
System.out.println(names);
names.set(3, "Can");       // replace index 3
System.out.println(names);
names.remove(4);           // remove index 4
System.out.println(names);
names.add(2, "Fatma");     // insert at index 2
System.out.println(names);
int index = names.indexOf("Banu");
System.out.println("index: " + index);
boolean hasEce = names.contains("Ece");
System.out.println("has Ece: " + hasEce);
System.out.println("size: " + names.size());
```

```
[Ali, Banu, Cem, Deniz, Ece]
[Ali, Banu, Cem, Can, Ece]
[Ali, Banu, Cem, Can]
[Ali, Banu, Fatma, Cem, Can]
index: 1
has Ece: false
size: 5
```

Table 5.4 gives the contents of the list and the new output at each step. The line numbers are the lines of the code above.

Step	Line	List	Variables	New output
1	1: `ArrayList<String> names = new ArrayList<>();`	empty	—	—
2	2: `names.add("Ali");`	Ali	—	—
3	3: `names.add("Banu");` 4: `names.add("Cem");` 5: `names.add("Deniz");` 6: `names.add("Ece");`	Ali, Banu, Cem, Deniz, Ece	—	—
4	7: `System.out.println(names);`	Ali, Banu, Cem, Deniz, Ece	—	`[Ali, Banu, Cem, Deniz, Ece]`

Step	Line	List	Variables	New output

5	8: `names.set(3, "Can"); // replace index 3`
	9: `System.out.println(names);`
		Ali, Banu, Cem, Can, Ece	—	`[Ali, Banu, Cem, Can, Ece]`

6	10: `names.remove(4); // remove index 4`
	11: `System.out.println(names);`
		Ali, Banu, Cem, Can	—	`[Ali, Banu, Cem, Can]`

7	12: `names.add(2, "Fatma"); // insert at index 2`
	13: `System.out.println(names);`
		Ali, Banu, Fatma, Cem, Can	—	`[Ali, Banu, Fatma, Cem, Can]`

8	14: `int index = names.indexOf("Banu");` 15: `System.out.println("index: " + index);`	Ali, Banu, Fatma, Cem, Can	index=1	`index: 1`
9	16: `boolean hasEce = names.contains("Ece");` 17: `System.out.println("has Ece: " + hasEce);`	Ali, Banu, Fatma, Cem, Can	index=1, hasEce=false	`has Ece: false`
10	18: `System.out.println("size: " + names.size());`	Ali, Banu, Fatma, Cem, Can	index=1, hasEce=false	`size: 5`

Table 5.4: Tracing the `ArrayList` methods. The List column gives the elements at the end of the step, in index order.

In the first step an empty list is created; `size()` is 0. `add("Ali")` puts the first element at index 0. Four more `add` calls place the elements at indices 1, 2, 3, and 4. When the list is printed, the elements appear in insertion order. Figure 5.5 shows three important moments of the trace.



Figure 5.5: Three moments of change. On the left, `set(3, "Can")` : Can is written in place of Deniz, and the size stays 5. In the middle, `remove(4)` : the last element, Ece, is deleted and no shifting is needed. On the right, `add(2, "Fatma")` : Cem and Can shift one place right and Fatma goes to index 2.

`set(3, "Can")` does not change the size; only the content of cell 3 changes. The method returns the old value ("Deniz"), but the return value is not used in this example. `remove(4)` deletes the last element. Since there is no element behind the last one, nothing shifts and the operation takes O(1). `add(2, "Fatma")` , on the other hand, inserts into the middle of the list. First Cem and Can at indices 2 and 3 are moved one cell to the right, then Fatma is written into the freed cell 2.

The last three lines perform searches. `indexOf("Banu")` scans the list from the start and returns the index of the first match: 1. `contains("Ece")` does the same scan, but because Ece was deleted earlier the result is `false` . The final size is 5: 5 additions and 1 insertion were made, and 1 element was deleted. Both search methods are O(n) because in the worst case they traverse the whole list. If the question "is it there?" is asked often on a list, switching to a set is the right choice.

5.2.4 Inserting into and deleting from the middle

To see the cost of shifting more closely, let us insert an element into the middle of a four-number list and delete an element from the front:

```
List<Integer> list2 = new ArrayList<>(Arrays.asList(10, 20, 30, 40));
list2.add(1, 15);      // insert: 20, 30, 40 shift right
System.out.println(list2);
list2.remove(0);       // remove first: all shift left
System.out.println(list2);
```

```
[10, 15, 20, 30, 40]
[15, 20, 30, 40]
```

The expression `new ArrayList<>(Arrays.asList(10, 20, 30, 40))` creates a modifiable list filled with the given values. This pattern is often used in short examples. Figure 5.6 shows the two operations step by step.

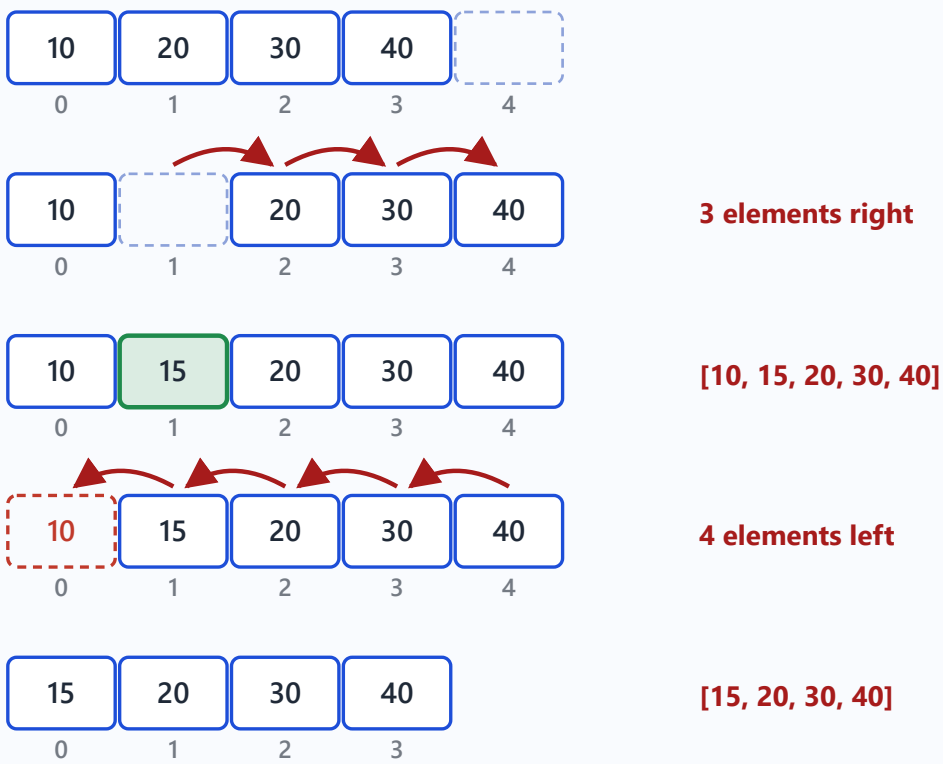


Figure 5.6: Insertion into the middle and removal from the front. Initially the list is `[10, 20, 30, 40]`. For `add(1, 15)`, the elements after index 1 first shift one place right, starting from the end, and then 15 is written into the freed cell. `remove(0)` deletes 10; the four elements behind it shift one place left. The result is `[15, 20, 30, 40]`.

The right shift starts from the end. First 40, then 30, then 20 move one cell to the right. Had it started from the front, 20 would overwrite 30 and 30 would be lost. In the four-element list, `add(1, 15)` moves three elements. In general, inserting at position `i` moves `n - i` elements. The worst case is insertion at the front; all `n` elements change places. The same holds for deletion: `remove(0)` pulls all the elements behind it one place to the left.

5.2.5 LinkedList

Definition 5.5

A **doubly linked list** is a structure that stores its elements in **nodes**. Each node holds a value and references to the previous (`prev`) and next (`next`) nodes. `LinkedList` uses this structure; it is derived from the `AbstractSequentialList` class and implements both the `List` and the `Deque` interface.

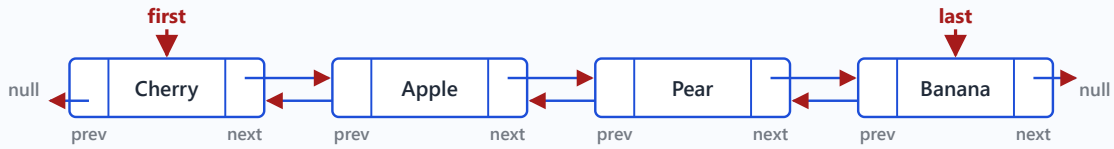


Figure 5.7: A doubly linked list: Cherry, Apple, Pear, Banana. `first` points to the first node and `last` to the last node. The `next` arrow of each node goes to the next node and the `prev` arrow to the previous node; the arrows at the ends are `null`.

In `LinkedList` there is no concept of capacity. Each addition creates a new node and links it to its neighbors. No element is shifted. That is why insertion and removal at the ends of the list are O(1): the `first` and `last` references keep the ends at hand, and only a few references change.

The price is paid in access by index. The nodes do not lie one after another in memory; each can be anywhere in the heap. To find element number 3, you must start from the front and follow the `next` links one by one. So `get(i)` is O(n). Java's implementation makes a small improvement: if the index is in the first half of the list it walks from the front, and if it is in the second half it walks from the back. This reduces the walk to at most `n/2` steps, but the growth rate is still O(n).

The analogy in the slides explains this trade-off well. A linked list is like a row of people holding hands. If someone is to step in between, only two people change hands. But to find the 3rd person, you must count from the start.

```
LinkedList<String> fruits = new LinkedList<>();
fruits.add("Apple");
fruits.add("Pear");
fruits.addFirst("Cherry");   // add to front
fruits.addLast("Banana");      // add to end
System.out.println(fruits);
System.out.println(fruits.getFirst());
System.out.println(fruits.getLast());
fruits.removeFirst();
System.out.println(fruits);
```

```
[Cherry, Apple, Pear, Banana]
Cherry
Banana
[Apple, Pear, Banana]
```

The methods `addFirst`, `addLast`, `getFirst`, `getLast`, `removeFirst`, and `removeLast` in this example come from the `Deque` interface. `add`, `get`, and `remove(i)` come from `List`. Because the variable is declared with type `LinkedList`, the methods of both interfaces can be used.

Tracing the nodes step by step

Table 5.5 traces the same program line by line. The Nodes column shows the order the list gives when traversed from `first` along the `next` links.

Step	Line	Nodes (front to back)	New output
1	1: `LinkedList<String> fruits = new LinkedList<>();`	first = last = null	—
2	2: `fruits.add("Apple");`	[Apple]	—
3	3: `fruits.add("Pear");`	[Apple, Pear]	—
4	4: `fruits.addFirst("Cherry");`	[Cherry, Apple, Pear]	—
5	5: `fruits.addLast("Banana");`	[Cherry, Apple, Pear, Banana]	—
6	6: `System.out.println(fruits);`	[Cherry, Apple, Pear, Banana]	`[Cherry, Apple, Pear, Banana]`
7	7: `System.out.println(fruits.getFirst());`	[Cherry, Apple, Pear, Banana]	`Cherry`
8	8: `System.out.println(fruits.getLast());`	[Cherry, Apple, Pear, Banana]	`Banana`
9	9: `fruits.removeFirst();`	[Apple, Pear, Banana]	—
10	10: `System.out.println(fruits);`	[Apple, Pear, Banana]	`[Apple, Pear, Banana]`

Table 5.5: Tracing the `LinkedList` operations.

Initially the list is empty: `first` and `last` are both `null`. For "Apple" the first node is created; since the single node is both first and last, both references point to it. When "Pear" is linked at the end, three references change. Apple's `next` points to Pear, Pear's `prev` points to Apple, and `last` moves to Pear. Figure 5.8 shows this linking and the effect of `removeFirst`.

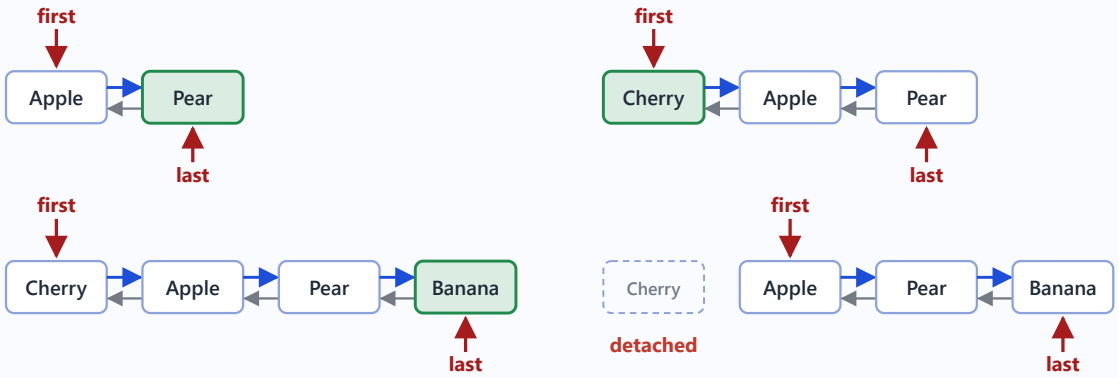


Figure 5.8: Four moments in the linked list. At top left, Pear is linked at the end. At top right, `addFirst("Cherry")` adds the new node at the front; Apple's `prev` and `first` change. At bottom left, `addLast("Banana")`. At bottom right, `removeFirst()`: `first` moves to Apple, Apple's `prev` becomes `null`, and the Cherry node is cut off from the list.

In the `addFirst("Cherry")` call, the new node's `next` points to Apple, Apple's `prev` points to Cherry, and `first` moves to Cherry. In an array, insertion at the front would shift all the elements. Here only three references changed. `addLast("Banana")` does the same job at the end. `println` prints the list by starting from `first` and following the `next` links.

`getFirst()` and `getLast()` return directly the value of the node that `first` and `last` point to; both are O(1). `removeFirst()` moves `first` to the next node and sets that node's `prev` to `null`. Since no reference to the Cherry node remains, the garbage collector cleans it up later.

On an empty `LinkedList`, `peekFirst()` returns `null`, while `getFirst()` throws a `NoSuchElementException`.

5.2.6 ArrayList or LinkedList?

The two classes implement the same `List` interface, but the costs of their operations are very different. Table 5.6 compares the basic operations.

Operation	`ArrayList`	`LinkedList`
`get(i)`, `set(i, e)`	O(1)	O(n): walks node by node
Append at the end `add(e)`	O(1) on average (occasional copying)	O(1)
Insert at front / remove from front	O(n): all elements shift	O(1)

Operation	`ArrayList`	`LinkedList`
Insert into / remove from the middle	O(n): shifting	O(n): finding the position + O(1) linking
Memory use	low: only the array	high: 2 extra references per node

Table 5.6: The costs of `ArrayList` and `LinkedList` operations.

The fourth row of the table is often misunderstood. People say "insertion into the middle of a linked list is O(1)", but this holds only if the right node is already in hand. The call `add(i, e)` first walks to find node number `i`; that walk is O(n). The linking step is O(1). In the end, insertion into the middle is O(n) in both structures. The difference is that `ArrayList` shifts elements while `LinkedList` counts nodes. Shifting an array that lies contiguously in memory is often faster than walking across scattered nodes.

To see how large the difference is, let us measure two extreme cases. The measurement program inserts 100,000 elements at the front of each list, then calls `get(i)` 100,000 times:

```
insert at front ms: ArrayList 539, LinkedList 1 | get(i) ms: ArrayList 0, LinkedList 5161
```

The times vary with the machine and the run; these numbers come from a run on the computer on which this note was written. The ratios, however, are stable. For insertion at the front, `LinkedList` is hundreds of times faster, because `ArrayList` shifts all the elements on every insertion and the total work grows to $O(n^2)$. For reading by index the situation reverses: `ArrayList` finishes in under a millisecond, while `LinkedList` spends seconds because it walks node by node for every `get`.

In most cases the right choice is `ArrayList`. Programs mostly fill a list by appending at the end and read it by index or with for-each. For these two jobs `ArrayList` is at least as fast as `LinkedList` and uses less memory. If insertions and removals are frequent and especially at the ends of the list, `LinkedList` can be considered. For queue and stack needs, however, `ArrayDeque` is usually a better choice.

Common mistake

Traversing a `LinkedList` with an indexed `for` loop. In the loop `for (int i = 0; i < l.size(); i++) l.get(i)`, every `get(i)` walks from the front or the back. In a list of `n` elements the total work becomes $O(n^2)$, and as the measurement showed, it takes seconds on large lists. A linked list is traversed with for-each or an iterator; these ways move on to the next node at each step, and the total work is O(n).

5.2.7 Vector and Stack

`Vector` and `Stack` are left over from Java 1.0; when the Collections Framework arrived with Java 1.2, these two classes were adapted to the `List` interface. They are still seen in old code and in some textbooks. For this reason you need to know how they work, but in new code more suitable classes are used in their place.

Vector

`Vector` is an expandable dynamic array that implements the `List` interface. It starts with the specified capacity, or with 10 cells if none is specified. When full, its capacity doubles; this is its difference from the 1.5-fold growth of `ArrayList`. The main distinction, however, is that its methods are **synchronized**. Every time a method is called, the object is locked. In this way several threads can access the same list at the same time safely. But the lock is acquired and released on every call even in single-threaded programs, and this wastes time for nothing.

```
Vector<Integer> v = new Vector<>();   // capacity: 10
for (int i = 1; i <= 11; i++) {
    v.addElement(i * 10);
}
System.out.println(v.size() + " / " + v.capacity());
System.out.println(v.elementAt(0));
v.removeElement(110);    // value 110
System.out.println(v.size() + " / " + v.capacity());
```

```
11 / 20
10
10 / 20
```

The values 10, 20, …, 110 are added. When the eleventh element arrives, the 10-cell array is full and the capacity doubles to 20. `elementAt(0)` returns the first element. `removeElement(110)` deletes the element whose value is 110; the size falls to 10, but the capacity stays at 20. Deletion does not shrink the capacity. Table 5.7 gives the old-style methods specific to `Vector`. All of these methods have an equivalent in `List`: `add` instead of `addElement`, `get` instead of `elementAt`.

Method	Purpose
`addElement(E obj)`	adds an element
`elementAt(int i)`	returns the element at `i`
`removeElement(Object obj)`	removes the element
`size()`	number of elements
`capacity()`	current capacity

Table 5.7: Methods specific to `Vector`.

Stack

Definition 5.6

A **stack** is a structure that works on the last-in, first-out (LIFO) principle. The most recently added element sits at the **top**, and the oldest at the **base**. Insertion and removal are done only at the top.

In everyday life a stack is like plates piled on top of each other: a plate is always put on top and taken from the top. In programming a stack has many uses. Undo in a text editor reverses the most recent action first. Whether the parentheses in an expression are balanced is checked with a stack. Depth-first search keeps the nodes to be visited on a stack. Method calls are also stored on the call stack.

Java's `Stack` class is derived from `Vector`. Table 5.8 gives its methods and Figure 5.9 its basic operation.

Method	Purpose
`push(E item)`	adds to the top
`pop()`	removes and returns the top element
`peek()`	returns the top element without removing it
`empty()`	is the stack empty?
`search(Object o)`	position counted from the top starting at 1; −1 if absent

Table 5.8: The methods of the `Stack` class.

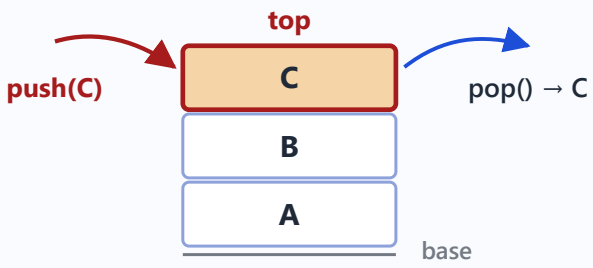


Figure 5.9: The stack. `push(C)` puts C on top; `pop()` removes and returns the top element C. A stays at the bottom (the base).

```
Stack<String> stack = new Stack<>();
stack.push("A");
stack.push("B");
stack.push("C");
System.out.println(stack);
System.out.println(stack.peek());
System.out.println(stack.pop());
System.out.println(stack.search("A"));
System.out.println(stack.empty());
```

```
[A, B, C]
C
C
2
false
```

Table 5.9 traces the program step by step. The Stack column gives the elements from bottom to top.

Step	Line	Variables	New output
1	1: `Stack<String> stack = new Stack<>();`	stack=[]	—
2	2: `stack.push("A");`	stack=[A]	—
3	3: `stack.push("B");`	stack=[A, B]	—
4	4: `stack.push("C");`	stack=[A, B, C]	—
5	5: `System.out.println(stack);`	stack=[A, B, C]	`[A, B, C]`
6	6: `System.out.println(stack.peek());`	stack=[A, B, C]	`C`
7	7: `System.out.println(stack.pop());`	stack=[A, B]	`C`
8	8: `System.out.println(stack.search("A"));`	stack=[A, B]	`2`
9	9: `System.out.println(stack.empty());`	stack=[A, B]	`false`

Table 5.9: Tracing the `Stack` operations.

After three `push` calls, A is at the bottom and C at the top. `println` prints the stack from bottom to top: `[A, B, C]`. This order may surprise at first, because the "top" is at the right end. Since `Stack` is a `Vector`, it keeps its elements in the internal array in insertion order, and the top is the last cell of the array. `peek()` returns C but does not remove it. `pop()` removes and returns C; the last one in has come out first. `search("A")` counts the position from the top starting at 1: B is 1, A is 2. This count differs from indices: it starts at 1, not 0. On the last line the stack holds two elements, and `empty()` returns `false`.

Since `push`, `pop`, and `peek` work at the end of the internal array, all three are O(1) (`push` amortized). If `pop()` or `peek()` is called on an empty stack, an `EmptyStackException` is thrown:

```
Exception in thread "main" java.util.EmptyStackException
```

In modern code, `ArrayDeque` (or `LinkedList`), which implements the `Deque` interface, is recommended in place of `Stack`. `Stack` has two problems. The first is the unnecessary synchronization inherited from `Vector`. The second is the extra methods that inheritance brings. Since `Stack` is a `Vector`, calls such as `add(0, x)` or `remove(1)` can also be made. These break the "access only from the top" rule of the stack.

5.2.8 Traversing a list

There are three common ways to process all the elements of a list in order:

```
List<String> courses = new ArrayList<>(
        Arrays.asList("Physics", "Java", "Circuits"));
// 1) indexed for
for (int i = 0; i < courses.size(); i++) {
    System.out.print(courses.get(i) + " ");
}
System.out.println();
// 2) enhanced for (for-each)
for (String d : courses) {
    System.out.print(d + " ");
}
System.out.println();
// 3) forEach + lambda (Java 8+)
courses.forEach(d -> System.out.print(d + " "));
System.out.println();
```

```
Physics Java Circuits
Physics Java Circuits
Physics Java Circuits
```

The first way is the indexed `for` loop. The loop variable `i` runs from 0 to `size() - 1` and each element is obtained with `get(i)`. This way is used when the index itself is needed, for example to print "Java is in 3rd place". It works only on lists, because sets and queues have no `get(i)`.

The second way is the enhanced `for`, that is, the for-each loop. It is the most readable form and works on every collection that is `Iterable`. The compiler turns this loop into an iterator behind the scenes. This transformation is traced in detail in the iterator section.

The third way is the `forEach` method and a lambda expression (Java 8+). The expression `d -> System.out.print(d + " ")` means "for each element `d`, do this". It is a short form, but it cannot modify local variables outside the loop body. If you try to increment a counter from inside the lambda, the compiler stops:

```
LambdaOutside.java:6: error: local variables referenced from a lambda expression must be
final or effectively final
        l.forEach(x -> total += x);
                       ^
1 error
```

A lambda can only read local variables from outside; those variables must be `final` or effectively `final` (assigned once and never changed). For accumulating values such as a total, either for-each is used or stream operations such as `sum` and `reduce`.

The output of the three ways is the same. If you need to remove elements from the list while traversing, none of the three is suitable. For this job the iterator's own `remove` method is used.

5.3 Queues

A queue holds elements waiting to be processed. Documents sent to a printer, requests arriving at a server, and moves waiting their turn in a game all wait in a queue. The real question of a queue is "who is next?" The answer depends on the structure: in an ordinary queue the first to arrive, in a priority queue the most urgent, and in a stack the most recent arrival is next. This section describes these three behaviors through `Queue`, `PriorityQueue`, and `Deque`, respectively.

5.3.1 The Queue interface

Definition 5.7

A **queue** is a structure that works on the first-in, first-out (FIFO) principle. A new element enters at the end of the queue; the element to be processed is taken from the front of the queue. Elements leave in the order in which they were added.

A supermarket checkout is a useful analogy. You join the line at the back, and the person at the front pays. No one can enter or leave in the middle of the line. Figure 5.10 shows this arrangement.

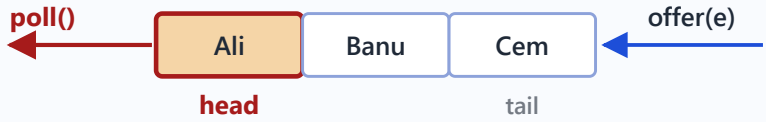


Figure 5.10: The queue. `offer(e)` adds at the tail, `poll()` removes the element at the head (Ali).

The `Queue` interface has three well-known implementations. `LinkedList` works with a doubly linked list and `ArrayDeque` with a dynamic array; both are true FIFO queues. `PriorityQueue`, on the other hand, removes elements by priority; therefore it is not FIFO. Typical uses of a queue are a print queue, a task queue, and breadth-first search.

Two forms of each operation

The `Queue` interface offers each basic operation in two forms. One form throws an exception on failure, the other returns a special value. Table 5.10 gives this pairing.

Operation	Throws an exception	Returns a special value
Insert (at the tail)	`add(e)`	`offer(e)` → `false`
Remove (from the head)	`remove()`	`poll()` → `null`
Examine (the head)	`element()`	`peek()` → `null`

Table 5.10: The two forms of the `Queue` methods.

When the queue is empty, `remove()` and `element()` throw `NoSuchElementException`, while `poll()` and `peek()` return `null`. On the insertion side, the difference appears only in queues with limited capacity. On a full queue, `add` throws an exception while `offer` returns `false`. Since `LinkedList` and `ArrayDeque` grow without limit, insertion always succeeds in these two. The difference in behavior on an empty queue is seen below:

```
import java.util.*;
public class EmptyQueue {
    public static void main(String[] args) {
        Queue<String> q = new LinkedList<>();
        System.out.println(q.poll() + " " + q.peek());
        try { q.element(); } catch (RuntimeException e) { System.out.println("element: " +
e); }
        q.remove();
    }
}
```

```
null null
element: java.util.NoSuchElementException
Exception in thread "main" java.util.NoSuchElementException
```

Tracing a queue

```
Queue<String> queue = new LinkedList<>();
queue.offer("Ali");
queue.offer("Banu");
queue.offer("Cem");
System.out.println(queue);
System.out.println(queue.peek());
System.out.println(queue.poll());
System.out.println(queue.poll());
System.out.println(queue.size());
System.out.println(queue.poll());
System.out.println(queue.poll());
```

```
[Ali, Banu, Cem]
Ali
Ali
Banu
1
Cem
null
```

The variable has type `Queue` and the object is a `LinkedList`. In this way only queue methods can be used; a call that would break the queue discipline, such as `queue.get(1)`, does not compile. Table 5.11 gives the contents of the queue at each step.

Step	Line	Queue (head to tail)	New output
1	1: `Queue<String> queue = new LinkedList<>();`	empty	—
2	2: `queue.offer("Ali");`	Ali	—
3	3: `queue.offer("Banu");`	Ali, Banu	—
4	4: `queue.offer("Cem");`	Ali, Banu, Cem	—
5	5: `System.out.println(queue);`	Ali, Banu, Cem	`[Ali, Banu, Cem]`
6	6: `System.out.println(queue.peek());`	Ali, Banu, Cem	`Ali`
7	7: `System.out.println(queue.poll());`	Banu, Cem	`Ali`
8	8: `System.out.println(queue.poll());`	Cem	`Banu`
9	9: `System.out.println(queue.size());`	Cem	`1`
10	10: `System.out.println(queue.poll());`	empty	`Cem`
11	11: `System.out.println(queue.poll());`	empty	`null`

Table 5.11: Tracing the `Queue` operations. The Queue column gives the elements from head to tail.

Three `offer` calls add Ali, Banu, and Cem to the tail in order. `peek()` returns the head (Ali) but does not change the queue. The first `poll()` removes and returns Ali; the first one in has come out first. The second `poll()` removes Banu. At this point `size()` is 1. The third `poll()` removes Cem and the queue becomes empty. The fourth `poll()` does not raise an error; it returns `null`. Since the head and tail of the queue are found in constant time, `offer`, `poll`, and `peek` are O(1).

5.3.2 PriorityQueue and the binary heap

Definition 5.8

A **priority queue** is a queue in which the element that leaves is always the highest-priority element; the order of insertion is not decisive. In `PriorityQueue`, the default priority is the natural order: the smallest element leaves first.

Typical uses of a priority queue are task scheduling, emergency-room order, and shortest-path algorithms. In an emergency room, patients are called not in order of arrival but according to the severity of their condition. The operating system also moves priority jobs to the front. The elements must be comparable. Either the element class implements the `Comparable` interface, or a `Comparator` is given to the queue. A comparator is also given for the reverse order, that is, for the largest to leave first. Table 5.12 lists the basic methods.

Method	Purpose	Cost
`add(E e)`	adds an element	O(log n)
`poll()`	removes and returns the highest-priority element	O(log n)
`peek()`	returns the highest-priority element without removing it	O(1)
`remove(Object o)`	removes the specified element	O(n)
`size()`	number of elements	O(1)

Table 5.12: `PriorityQueue` methods and their costs.

`PriorityQueue` internally uses a **binary heap**. This word "heap" shares its name with the memory heap, but it is a different structure. A binary heap is a complete binary tree stored in an array. The rule of the tree is this: every parent is less than or equal to its children. This rule holds only between parent and child; there is no order among siblings. As a consequence of the rule, the smallest element is always at the root.

The tree is stored in an array as follows. The root is at index 0. The children of the node at index `i` are at `2i + 1` and `2i + 2`, and its parent is at `(i - 1) / 2`. In this way you can move up and down the tree without keeping any references. Figure 5.11 shows a four-element heap both as a tree and as an array.

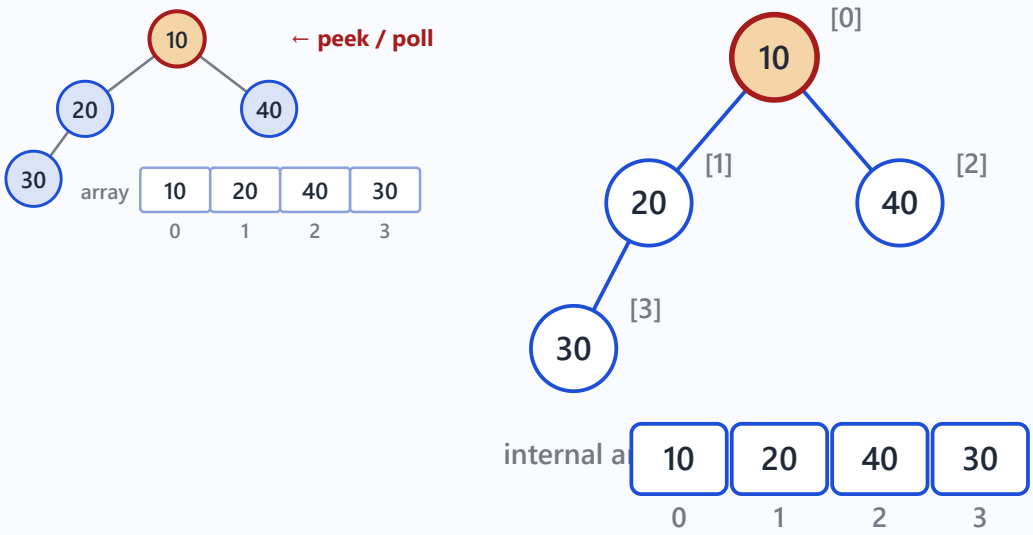


Figure 5.11: The binary heap. On the left, the root is the smallest element; `peek` and `poll` use the root. On the right is the internal layout after 30, 10, 40, 20 are added: the array `[10, 20, 40, 30]`. In the tree, 10 is the root; 20 and 40 are its children, and 30 is the child of 20.

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.add(30);
pq.add(10);
pq.add(40);
pq.add(20);
System.out.println(pq.peek());   // smallest
System.out.println(pq);          // heap order
while (!pq.isEmpty()) {
    System.out.print(pq.poll() + " ");
}
System.out.println();
```

```
10
[10, 20, 40, 30]
10 20 30 40
```

The second line of the output often surprises. `println(pq)` does not print the elements in sorted order; it prints the internal array as it is. The internal array obeys only the rule "every parent is less than its children": 10 is less than its children 20 and 40, and 20 is less than its child 30. But there is no order between 40 and 30. For a sorted result, the elements are removed one by one with `poll`; the removal order is always from smallest to largest.

Tracing the heap rule step by step

Table 5.13 shows what each insertion and removal does internally. The Internal array column gives the array after each step; the values were verified with a version of the same program that prints the internal array at each step.

Step	Line	Internal array	Variables	New output
1	1: `PriorityQueue<Integer> pq = new PriorityQueue<>();`	[]	—	—
2	2: `pq.add(30);`	[30]	—	—
3	3: `pq.add(10);`	[10, 30]	—	—
4	4: `pq.add(40);`	[10, 30, 40]	—	—
5	5: `pq.add(20);`	[10, 20, 40, 30]	—	—
6	6: `System.out.println(pq.peek());`	[10, 20, 40, 30]	—	`10`
7	7: `System.out.println(pq);`	[10, 20, 40, 30]	—	`[10, 20, 40, 30]`
8	9: `System.out.print(pq.poll() + " ");`	[20, 30, 40]	x=10	`10`
9	9: `System.out.print(pq.poll() + " ");`	[30, 40]	x=20	`20`
10	9: `System.out.print(pq.poll() + " ");`	[40]	x=30	`30`
11	9: `System.out.print(pq.poll() + " ");`	[]	x=40	`40`
12	11: `System.out.println();`	[]	—	—

Table 5.13: Tracing the internal array of `PriorityQueue`.

Insertion is always done at the end of the array, and then the new element **sifts up** as far as necessary. 30 enters the empty array and becomes the root. 10 is added at index 1. Its parent is 30 at index 0, and 10 is less than it; since the rule is broken, the two swap. The root is now 10. 40 enters at index 2; since it is greater than its parent 10, it stays in place. 20 enters at index 3. Its parent is 30 at index 1. Since 20 is smaller, they swap. 20 is now at index 1; since it is greater than its parent 10, it stops. Figure 5.12 shows these steps on the tree.

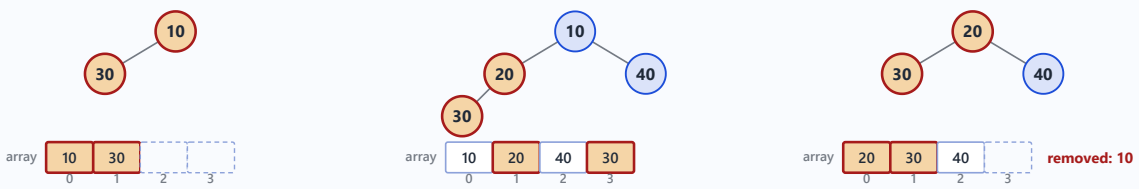


Figure 5.12: Three moments in the binary heap. On the left, 10 has been added and has risen to the root by swapping with 30. In the middle, 20 has been added, swapped with 30, and stopped below 10. On the right, the first `poll()`: the root 10 is removed, the last element 30 comes to the root, and it sinks by swapping with its smaller child 20.

`peek()` looks only at the root and is O(1). Removal is done as follows. The root is removed, the last element of the array is moved to the root, and it **sifts down** as far as necessary. In the first `poll()`, 10 is removed and the last element 30 comes to the root. The children of 30 are 20 and 40; it swaps with the smaller one, 20. The array becomes `[20, 30, 40]`. In the second `poll()`, 20 is removed, 40 comes to the root and swaps with its smaller child 30: `[30, 40]`. Then 30 and 40 are removed in turn and the queue becomes empty. The removal order is 10 20 30 40.

Sifting up and sifting down follow only a single branch. The height of a complete binary tree with `n` elements is about $\log_2 n$. That is why `add` and `poll` are O(log n). Even in a heap of a million elements, each operation makes at most about 20 comparisons. `remove(Object o)`, however, first scans the array to find the element and is O(n).

> **Common mistake**
>
> **Traversing a priority queue with for-each and expecting a sorted result.** `for (int x : pq)` and `println(pq)` give the internal array in array order; the result is not sorted. To take the elements in priority order, the pattern `while (!pq.isEmpty()) pq.poll()` is used. This pattern empties the queue; if the queue is needed again afterwards, a copy is taken first.

5.3.3 Defining the priority

Numbers and strings are already comparable; their natural order is defined. Objects of our own classes, on the other hand, can enter a priority queue only if an order is defined. The first way to do this is for the class to implement the `Comparable` interface:

```
class Task implements Comparable<Task> {
    String name;
    int priority;          // smaller = more urgent

    Task(String name, int priority) {
        this.name = name;
        this.priority = priority;
    }

    @Override
    public int compareTo(Task o) {
        return Integer.compare(priority, o.priority);
    }

    @Override
    public String toString() {
        return name + "(" + priority + ")";
    }
}
```

```
PriorityQueue<Task> tasks =
        new PriorityQueue<>();
tasks.add(new Task("Report", 3));
tasks.add(new Task("Outage", 1));
tasks.add(new Task("Meeting", 2));
while (!tasks.isEmpty()) {
    System.out.println(tasks.poll());
}
```

```
Outage(1)
Meeting(2)
Report(3)
```

The `Task` class implements `Comparable<Task>` and writes a single method: `compareTo`. This method compares two tasks. It returns a negative number if `this` should come first, 0 if they are equal, and a positive number if it should come later. `Integer.compare(priority, o.priority)` does exactly this. Here a small priority number means urgent. Outage (1) leaves first, Meeting (2) next, and Report (3) last. The order of insertion plays no role at all.

If the reverse of the natural order is needed, there is no need to change the class. A comparator is passed to the queue's constructor. `Collections.reverseOrder()` is a ready-made comparator that reverses the natural order:

```
PriorityQueue<Integer> maxFirst =
        new PriorityQueue<>(
        Collections.reverseOrder());
maxFirst.add(30);
maxFirst.add(10);
maxFirst.add(40);
System.out.println(maxFirst.poll());
```

```
40
```

In this queue the largest element is at the root and leaves first. Comparators are treated in detail in the section on algorithms.

If the element class neither implements `Comparable` nor a comparator is given to the queue, the program compiles but a `ClassCastException` is thrown on the first insertion.

5.3.4 Deque and ArrayDeque

Definition 5.9

A **double-ended queue** (`Deque` , pronounced "deck") is a queue in which elements can be added and removed from both the front and the back. It combines the functions of a queue (FIFO) and a stack (LIFO) in a single structure.



Figure 5.13: The double-ended queue. At the front, `addFirst` / `offerFirst` insert and `removeFirst` / `pollFirst` remove; at the back, `addLast` / `offerLast` and `removeLast` / `pollLast` do the same jobs.

The `Deque` interface has two implementations: `LinkedList` (a doubly linked list) and `ArrayDeque` (a dynamic array). There are the same three operations for both ends: insert, remove, and examine. Table 5.14 collects these methods. As in `Queue` , each operation has two forms, one that throws an exception and one that returns a special value. `remove…` and `get…` throw an exception on an empty deque; `poll…` and `peek…` return `null` .

Operation	At the front	At the back
Insert	`addFirst(e)` (`offerFirst`)	`addLast(e)` (`offerLast`)
Remove and return	`removeFirst()` (`pollFirst`)	`removeLast()` (`pollLast`)
Return, do not remove	`getFirst()` (`peekFirst`)	`getLast()` (`peekLast`)

Table 5.14: The `Deque` methods. The forms in parentheses return `null` instead of throwing an exception on an empty deque.

`Deque` also carries the familiar names of the stack and queue interfaces. The stack operations work at the front: `push` = `addFirst` , `pop` = `removeFirst` , `peek` = `peekFirst` . Among the queue operations, `offer` adds at the back (`offerLast`) and `poll` takes from the front (`pollFirst`). If you keep these correspondences in mind, the contents of a deque are easy to trace.

ArrayDeque

`ArrayDeque` implements the double-ended queue with a **circular** array. The array keeps two indices: the head (`head`) and the tail (`tail`). Adding at the front moves the head index back by one, and adding at the back moves the tail index forward by one. If an index runs off one end of the array, it

wraps around to the other end. In this way, insertion and removal at both ends are O(1) with no shifting of elements. When the array is full, it is copied into a larger array, just as in `ArrayList`. Figure 5.14 shows this arrangement.



Figure 5.14: The circular array of `ArrayDeque`. The head index is at 6, and the last element is at index 0. Reading from head to tail visits cells 6 and 7 and then, wrapping around to the start, cell 0: the deque is 1, 5, 10.

In most cases `ArrayDeque` is both faster and lighter than `Stack` and `LinkedList`. Unlike `Stack`, it does not acquire a lock on every call. Unlike `LinkedList`, it does not create a separate node object for each element. For this reason the Java documentation recommends `ArrayDeque` for stack and queue needs. Its only restriction is that it does not accept `null` elements. Because `poll` returns `null` on an empty deque, a `null` element would be confused with this return value; the class therefore rejects `null` from the start:

```
import java.util.*;
public class NullDeque {
    public static void main(String[] args) {
        Deque<String> d = new ArrayDeque<>();
        d.offer(null);
    }
}
```

```
Exception in thread "main" java.lang.NullPointerException
```

To use it as a stack, the variable is declared with type `Deque`:

```
Deque<String> stack = new ArrayDeque<>();
stack.push("A");     // adds at the front
stack.push("B");
System.out.println(stack.pop());
```

```
B
```

A and B are pushed to the front; B is on top and `pop()` returns it. `Stack` prints from bottom to top; `ArrayDeque` prints from front to back, that is, from top to bottom: after the same pushes of A, B, C, one gives `[A, B, C]` and the other `[C, B, A]`.

Tracing an ArrayDeque

```
ArrayDeque<Integer> numbers = new ArrayDeque<>();
numbers.addFirst(5);
numbers.addLast(10);
numbers.addFirst(1);
System.out.println(numbers);
int firstNum = numbers.pollFirst();
int lastNum = numbers.pollLast();
System.out.println(firstNum + " " + lastNum);
System.out.println(numbers.peekFirst());
```

```
[1, 5, 10]
1 10
5
```

Table 5.15 traces the program step by step.

Step	Line	Deque (front to back)	Variables	New output
1	1: `ArrayDeque<Integer> numbers = new ArrayDeque<>();`	empty	—	—
2	2: `numbers.addFirst(5);`	5	—	—
3	3: `numbers.addLast(10);`	5, 10	—	—
4	4: `numbers.addFirst(1);`	1, 5, 10	—	—
5	5: `System.out.println(numbers);`	1, 5, 10	—	`[1, 5, 10]`
6	6: `int firstNum = numbers.pollFirst();`	5, 10	firstNum=1	—
7	7: `int lastNum = numbers.pollLast();`	5	firstNum=1, lastNum=10	—
8	8: `System.out.println(firstNum + " " + lastNum);`	5	firstNum=1, lastNum=10	`1 10`
9	9: `System.out.println(numbers.peekFirst());`	5	firstNum=1, lastNum=10	`5`

Table 5.15: Tracing the `ArrayDeque` operations. The Deque column gives the elements from front to back.

`addFirst(5)` adds the single element; 5 is both the front and the back. `addLast(10)` adds 10 at the back. `addFirst(1)` adds 1 at the front; 5 and 10 stay where they are, and no element shifts. The deque is printed from front to back as `[1, 5, 10]`. `pollFirst()` removes the 1 at the front. The

method returns an `Integer`; when it is assigned to `int firstNum`, automatic unboxing takes place. `pollLast()` removes the 10 at the back. Two values are printed: `1 10`. Finally, `peekFirst()` returns the 5 at the front but does not remove it.

This program has a subtle trap. `pollFirst()` returns `null` on an empty deque. If the result is assigned directly to an `int`, unboxing is performed on `null` and the program crashes:

```
import java.util.*;
public class EmptyDeque {
    public static void main(String[] args) {
        ArrayDeque<Integer> d = new ArrayDeque<>();
        int x = d.pollFirst();
    }
}
```

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"java.lang.Integer.intValue()" because the return value of
"java.util.ArrayDeque.pollFirst()" is null
```

The safety of the "special value" form is lost here. When using the `poll` family, the return value is first received as an `Integer` and checked for `null`, or `isEmpty()` is asked before the call.

5.4 Sets

The course codes a student has taken, the distinct words in a text, the visited nodes in a network: what these have in common is that repetition is meaningless. The same course is not taken twice, and the same word does not increase the number of distinct words. The set is designed for this kind of data. The real strength of a set is that it answers the question "is this element present?" very quickly. This section covers the contract of the set, how the hash table works, the `equals` and `hashCode` rules needed for our own classes to behave correctly in a set, and the sorted set.

5.4.1 The Set interface

Definition 5.10

A **set** is a collection that does not allow duplicate elements. It corresponds to the notion of a set in mathematics: only one of each element is present. When an attempt is made to add an element that already exists, `add` does not add it and returns `false`.

The `Set` interface in general gives no ordering guarantee; the traversal order depends on the implementation. The three implementations offer three different orders. `HashSet` is unordered. `LinkedHashSet` preserves insertion order. `TreeSet` keeps the elements sorted; the order is the natural order or a given `Comparator`. `Set` adds no new methods to `Collection`; it implements the same methods with the "no duplicates" rule.

```
Set<String> fruits = new HashSet<>();
System.out.println(fruits.add("Apple"));
System.out.println(fruits.add("Pear"));
System.out.println(fruits.add("Apple"));
System.out.println(fruits.size());
System.out.println(fruits.contains("Pear"));
fruits.remove("Pear");
System.out.println(fruits);
```

```
true
true
false
2
true
[Apple]
```

The first two `add` calls return `true` because they add new elements. The third, `add("Apple")`, tries to add an existing element; the set does not change and the result is `false`. That is why `size()` is 2, not 3. `contains("Pear")` returns `true`. After Pear is removed, only Apple remains in the set. The return value of `add` is useful in practice, too: the question "have we seen this word before?" is answered with a single `add`, without a separate `contains` call.

5.4.2 HashSet and the hash table

Definition 5.11

A **hash table** is a structure that distributes elements into cells called **buckets**. The `hashCode()` method of each element produces an integer; a bucket number is computed from this integer, and the element goes directly to that bucket. Insertion and lookup take O(1) on average.

`HashSet` stores unique elements without order. It is derived from `AbstractSet` and implements the `Set` interface. Internally it actually uses a `HashMap`: the elements of the set are the keys of the map. To search for an element, there is no need to scan from start to finish as in a list. The element's bucket is computed and only that bucket is examined.

The bucket number is found in two steps. First the element's `hashCode()` value is taken. Java mixes the upper 16 bits of this value with its lower 16 bits: `h ^ (h >>> 16)`. Then the last bits of the result become the bucket number. In a table with 16 buckets, the last 4 bits are used: `(h ^ (h >>> 16)) & 15`. This note's measurement program computes the buckets of four words with this formula:

```
salt hashCode=3522646 bucket=3
egg hashCode=100357 bucket=4
milk hashCode=3351579 bucket=8
coffee hashCode=-1355030580 bucket=7
Zeynep hashCode=-1621354421 bucket=7
Ahmet hashCode=63235125 bucket=1
Mert hashCode=2394618 bucket=14
Apple hashCode=63476538 bucket=2
Pear hashCode=2483462 bucket=3
```

The first four lines confirm the bucket numbers in the slides: salt 3, egg 4, milk 8, coffee 7. The same formula will also be used in the phone-book example in the section on maps. Figure 5.15 shows how these elements are placed in the table.

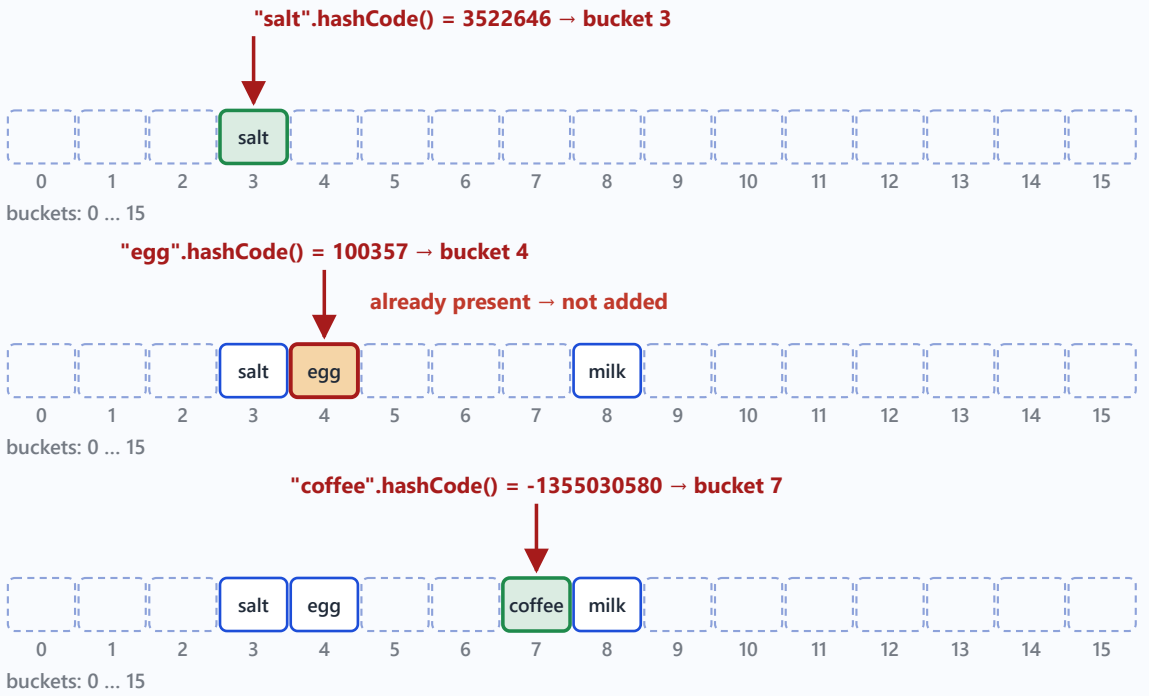


Figure 5.15: A `HashSet` with 16 buckets. At the top, `add("salt")`: bucket 3 is computed from the hash value. In the middle, the second `add("egg")`: bucket 4 already holds an element equal under `equals`, so the element is not added and `add` returns `false`. At the bottom, `add("coffee")` is placed in bucket 7; traversal follows the order of the buckets: `[salt, egg, coffee, milk]`.

The second `add("egg")` call shows how the "no duplicates" rule of the set is enforced. The element's bucket is computed: 4. It is compared with the elements in that bucket using `equals`. Since an equal element is found, no insertion is made. The whole set is not scanned; only a single bucket is examined.

The traversal order is the order of the buckets. The table is scanned from 0 to 15 and the elements in the filled buckets are given in turn. For this reason the order of a `HashSet` has nothing to do with insertion order. If the number of buckets changes, the order may change too. No program should rely on this order.

Two different elements can end up with the same bucket number; this is called a **collision**. For example, in the output above, coffee and Zeynep both have bucket 7. Colliding elements form a small chain in the same bucket, and the search in that chain is done with `equals`. Since Java 8, if the chain in a bucket grows too long (more than 8 elements and the table large enough), the chain is converted into a balanced tree. When the number of elements in the table exceeds 75% of the number of buckets (load factor 0.75), the table doubles in size and the elements are redistributed into new buckets. In this way the chains stay short and the average cost is O(1).

5.4.3 LinkedHashSet and the order of the three sets

`LinkedHashSet` is derived from `HashSet` and additionally links its elements together with a linked list. The hash table provides fast access and uniqueness, while the linked list preserves insertion order. If both speed and order matter, `LinkedHashSet` is chosen. If order does not matter and speed is critical, `HashSet` is enough. Its basic methods are the same: `add`, `remove`, `contains`, `size`, `clear`.

```
LinkedHashSet<Integer> numbers =
        new LinkedHashSet<>();
numbers.add(5);
numbers.add(10);
numbers.add(5);   // duplicate: not added
numbers.add(1);
System.out.println(numbers);
```

```
[5, 10, 1]
```

The second `add(5)` is not added because it is a duplicate. The following 1 is linked at the end of the list. The output is the insertion order: first 5, then 10, then 1. The difference is clearly visible when the same words are added to all three sets:

```
String[] words = {"salt", "egg", "milk", "egg", "coffee"};
Set<String> h = new HashSet<>();
Set<String> l = new LinkedHashSet<>();
Set<String> t = new TreeSet<>();
for (String k : words) {
    h.add(k);
    l.add(k);
    t.add(k);
}
System.out.println("HashSet:       " + h);
System.out.println("LinkedHashSet: " + l);
System.out.println("TreeSet:       " + t);
```

```
HashSet:       [salt, egg, coffee, milk]
LinkedHashSet: [salt, egg, milk, coffee]
TreeSet:       [coffee, egg, milk, salt]
```

One of the five words is a repeat (the second "egg"); all three sets have four elements. `HashSet` gives the order of the buckets: salt 3, egg 4, coffee 7, milk 8. `LinkedHashSet` preserves insertion order. `TreeSet` gives alphabetical order. Table 5.16 compares the three classes.

Class	Traversal order	Underlying structure	`add` / `contains`
`HashSet`	no guarantee (bucket order)	hash table	O(1) average
`LinkedHashSet`	insertion order	hash table + linked list	O(1) average
`TreeSet`	sorted (ascending)	red-black tree	O(log n)

Table 5.16: A comparison of the three set classes.

5.4.4 Our own classes: equals and hashCode

Sets of strings and numbers work as expected, because the `String` and `Integer` classes write the `equals` and `hashCode` methods by value. If these methods are not written in our own classes, the versions inherited from `Object` are used. The `equals` of `Object` checks whether two references point to the same object. Two separate objects with the same field values are therefore counted as "different".

```
class Point {
    int x, y;
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Point)) return false;
        Point n = (Point) o;
        return x == n.x && y == n.y;
    }
    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }
}
```

```
Set<PointA> s1 = new HashSet<>();
s1.add(new PointA(1, 2));
s1.add(new PointA(1, 2));
System.out.println(s1.size());

Set<Point> s2 = new HashSet<>();
s2.add(new Point(1, 2));
s2.add(new Point(1, 2));
System.out.println(s2.size());
```

```
2
1
```

`PointA` is the same two-field class, but `equals` and `hashCode` are not written. When `new PointA(1, 2)` is executed twice, two separate objects are created in memory. The `equals` of `Object` counts them as different, and the set size is 2. In the `Point` class, on the other hand, both methods are overridden. `equals` compares the fields. `hashCode` produces a number from the same fields with `Objects.hash(x, y)`. The same point enters the set once and the size is 1.

Definition 5.12

The `equals` – `hashCode` contract: if `a.equals(b)` is true, then `a.hashCode() == b.hashCode()` must also be true. The converse is not required: two objects with equal hash codes can be different (a collision). Therefore every class that overrides `equals` also overrides `hashCode`, and both are computed from the same fields.

The reason for the rule lies in the way the hash table works. First the bucket is found with `hashCode`, then the elements in that bucket are compared with `equals`. If two equal objects have different hash codes, they go to different buckets. `equals` is never called and both enter the set. A class that writes only `equals` and forgets `hashCode` makes exactly this mistake:

```
// only equals written, hashCode forgotten
class PointE {
    int x, y;
    PointE(int x, int y) { this.x = x; this.y = y; }
    @Override
    public boolean equals(Object o) {
        if (!(o instanceof PointE)) return false;
        PointE n = (PointE) o;
        return x == n.x && y == n.y;
    }
}
```

```
import java.util.*;
public class EqualsOnly {
    public static void main(String[] args) {
        PointE a = new PointE(1, 2), b = new PointE(1, 2);
        Set<PointE> s = new HashSet<>();
        s.add(a);
        s.add(b);
        System.out.println(a.equals(b) + " " + s.size() + " " + s.contains(new PointE(1,
2)));
        List<PointE> l = new ArrayList<>();
        l.add(a);
        System.out.println(l.contains(new PointE(1, 2)));
    }
}
```

```
true 2 false
true
```

The first line shows the contradiction: `a.equals(b)` is `true`, yet the set holds two elements. A `contains` with a new `PointE(1, 2)` also returns `false`, because the new object's hash code (and therefore its bucket) is different. On the second line, the same object is searched for in a list and found. `ArrayList.contains` uses only `equals`; it does not look at the hash code. The same class works correctly in a list and incorrectly in a set. The compiler notices this situation and, when compiled with the `-Xlint` option, gives a warning: `Class PointE overrides equals, but neither it nor any superclass overrides hashCode method`.

The number produced by `Objects.hash(x, y)` comes from a simple formula: `31 · (31 · 1 + x) + y`. For the point (1, 2) this gives 31 · 32 + 2 = 994. This number will be used in the next example.

5.4.5 Modifying an object in a set

Even when `equals` and `hashCode` are written correctly, there is one more trap. What happens if a field that determines the hash code is changed after the object has entered the set? The following program tries this:

```
Set<Point> set = new HashSet<>();
Point p = new Point(1, 2);
set.add(p);
System.out.println(set.contains(p));
p.x = 5;              // field changed
System.out.println(set.contains(p));
System.out.println(set.contains(new Point(1, 2)));
set.add(p);
System.out.println(set.size());
```

```
true
false
false
2
```

Table 5.17 traces the program step by step. In the Variables column, `hash` is the current hash code of `p`, and `bucket` is the bucket number computed from this code.

Step	Line	Variables	New output
1	1: `Set<Point> set = new HashSet<>();`	—	—
2	2: `Point p = new Point(1, 2);`	p=(1, 2), hash=994, bucket=2	—
3	3: `set.add(p);`	p=(1, 2), hash=994, bucket=2	—
4	4: `System.out.println(set.contains(p));`	p=(1, 2), hash=994, bucket=2	`true`
5	5: `p.x = 5; // field changed`	p=(5, 2), hash=1118, bucket=14	—
6	6: `System.out.println(set.contains(p));`	p=(5, 2), hash=1118, bucket=14	`false`
7	7: `System.out.println(set.contains(new Point(1, 2)));`	p=(5, 2), hash=1118, bucket=14, new=(1, 2)	`false`
8	8: `set.add(p);`	p=(5, 2), hash=1118, bucket=14	—
9	9: `System.out.println(set.size());`	p=(5, 2), hash=1118, bucket=14	`2`

Table 5.17: What happens when a field of an object in a set changes. The bucket number is the last 4 bits of the hash code (a table with 16 buckets).

Initially `Objects.hash(1, 2)` is 994. The binary representation of 994 is `1111100010`, and its last 4 bits are `0010`, that is, 2. `add(p)` puts the object into bucket 2. `contains(p)` again looks at bucket 2, `equals` returns true, and the result is `true`.

Then the assignment `p.x = 5` is made. The object's hash code is now `31 · 36 + 2 = 1118` and its bucket is 14. But the object still sits in bucket 2; the set is unaware that the field has changed. Figure 5.16 shows this situation and its consequence.

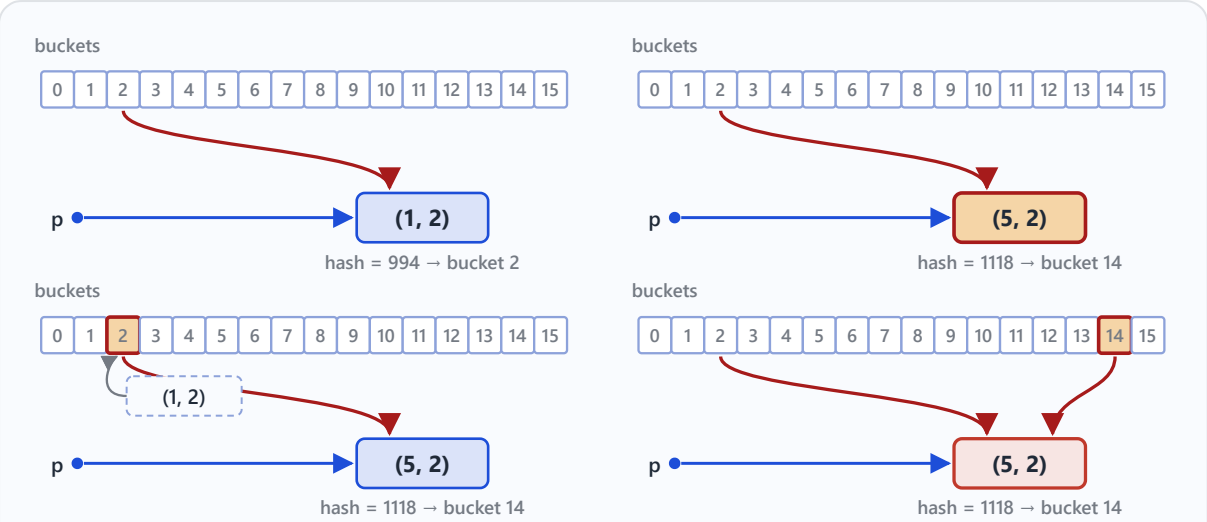


Figure 5.16: The lost element. At top left, `p` (1, 2) is placed in bucket 2. At top right, after `p.x = 5` the hash is 1118 and the bucket 14; the object stays in bucket 2. At bottom left, `new Point(1, 2)` looks at bucket 2, but the object there is now (5, 2) and `equals` returns false. At bottom right, `add(p)` finds bucket 14 empty and adds the same object a second time.

The second `contains(p)` looks at bucket 14. It is empty and the result is `false`. The object is in the set, but cannot be found. In the third query, a new `Point(1, 2)` is searched for. This object's bucket is 2 and there really is an object there. But that object is now (5, 2); `equals` returns false and the result is again `false`. Finally, `add(p)` finds bucket 14 empty and adds the same object a second time. `size()` becomes 2. The set contains the same object twice; the "no duplicates" rule has been broken.

The lesson of this bug is that no exception is thrown. The program keeps running and produces a wrong result. The rule is simple: the fields that enter `equals` and `hashCode` of an object in a hashed collection (`HashSet`, the key of a `HashMap`) are not modified. Immutable objects such as `String` and `Integer`, or classes whose fields are `final`, are chosen as keys. If a field has to change, the object is first removed from the set, modified, and then added again.

5.4.6 SortedSet and TreeSet

`SortedSet` is the set interface that stores its elements in sorted order. The order follows the natural order (`Comparable`) or a given `Comparator`. The interface offers new queries made possible by sortedness: the smallest and largest element, those smaller than a given value, and those within a given range. `TreeSet` is derived from `AbstractSet` and implements the `NavigableSet` (and hence `SortedSet`) interface. Internally it uses a **red-black tree**.

A red-black tree is a self-balancing binary search tree. The elements to the left of each node are smaller than it, those to the right are larger. During insertion and deletion, the tree rotates nodes so that its height stays on the order of log n. That is why insertion, deletion, and search are O(log n). It is slower than the O(1) of `HashSet`, but in return it gives order. Table 5.18 gives the sorted queries of `TreeSet` and Figure 5.17 the tree.

Method	Purpose
`first()` / `last()`	the smallest / the largest element
`headSet(to)`	those smaller than `to` (`to` excluded)
`tailSet(from)`	`from` and larger (`from` included)
`subSet(from, to)`	`from` included, `to` excluded
`comparator()`	the `Comparator` that orders the set; `null` for the natural order
`add`, `remove`	insertion, removal

Table 5.18: The sorted-set methods of `TreeSet`.

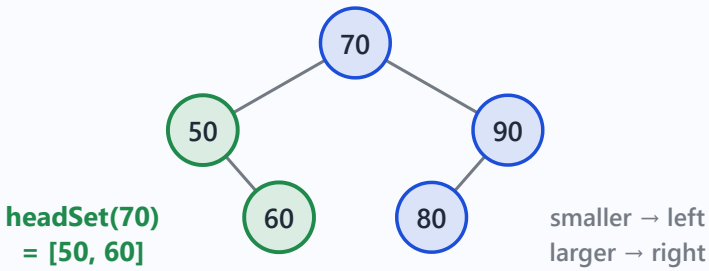


Figure 5.17: The tree inside a `TreeSet` (50, 60, 70, 80, 90). The root is 70; smaller elements are on the left, larger ones on the right. The green nodes show the range `headSet(70)` = `[50, 60]`.

```
TreeSet<Integer> grades = new TreeSet<>();
grades.add(70);
grades.add(50);
grades.add(90);
grades.add(60);
grades.add(80);
System.out.println(grades);
System.out.println(grades.first() + " " + grades.last());
System.out.println(grades.headSet(70));     // 70 excluded
System.out.println(grades.tailSet(70));     // 70 included
System.out.println(grades.subSet(60, 90));  // [60, 90)
System.out.println(grades.comparator());    // natural order
```

```
TreeSet<String> fruits = new TreeSet<>();
fruits.add("Pear");
fruits.add("Apple");
System.out.println(fruits);
```

```
[50, 60, 70, 80, 90]
50 90
[50, 60]
[70, 80, 90]
[60, 70, 80]
null
[Apple, Pear]
```

The grades are added in the order 70, 50, 90, 60, 80, but the set is always printed sorted: `[50, 60, 70, 80, 90]`. `first()` is 50 and `last()` is 90. In the range methods it matters whether the bounds are included. The rule is "start included, end excluded"; this is consistent with other range methods of Java such as `substring`. `headSet(70)` does not include 70: `[50, 60]`. `tailSet(70)` includes 70: `[70, 80, 90]`. `subSet(60, 90)` is the mathematical interval [60, 90): `[60, 70, 80]`. Because the natural order is used, `comparator()` returns `null`. Strings are also sorted by character codes: "Apple" comes before "Pear".

5.4.7 Set operations

Each of the mathematical operations of union, intersection, and difference has a counterpart in `Collection`. Table 5.19 gives the correspondence.

Operation	Method
union ∪	`addAll`
intersection ∩	`retainAll`
difference −	`removeAll`
subset ⊆	`containsAll`

Table 5.19: Set operations and Java methods.

```
Set<Integer> a = new TreeSet<>(Arrays.asList(1, 2, 3, 4));
Set<Integer> b = new TreeSet<>(Arrays.asList(3, 4, 5));

Set<Integer> union = new TreeSet<>(a);
union.addAll(b);
Set<Integer> inter = new TreeSet<>(a);
inter.retainAll(b);
Set<Integer> diff = new TreeSet<>(a);
diff.removeAll(b);

System.out.println("A ∪ B = " + union);
System.out.println("A ∩ B = " + inter);
System.out.println("A − B = " + diff);
System.out.println(a.containsAll(inter));
```

```
A ∪ B = [1, 2, 3, 4, 5]
A ∩ B = [3, 4]
A − B = [1, 2]
true
```

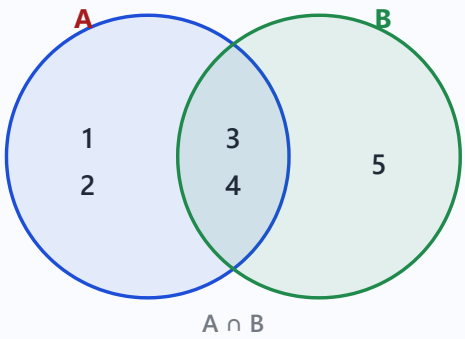


Figure 5.18: A Venn diagram. A = {1, 2, 3, 4}, B = {3, 4, 5}; the intersection is {3, 4}.

These methods modify the set on which they are called. Had `a.addAll(b)` been written, `a` itself would turn into the union and the original `a` would be lost in later operations. So a copy is made with `new TreeSet<>(a)` before each operation. The union adds the elements of `b` to the copy; the repeated 3 and 4 are not added a second time. The intersection leaves in the copy only the elements that are also in `b`. The difference removes the elements of `b` from the copy. The last line confirms that `inter` is a subset of `a`. Because `TreeSet` is used, the results are printed sorted. These operations perform a `contains` or `add` for each element of the collection given as a parameter; their cost is the number of elements times the cost of that operation.

5.5 Maps

Getting from a student number to a student, from a product name to a price, from a word to its occurrence count is among the most frequent jobs in programming. In these jobs the data comes in pairs: a key and a value bound to it. A map stores these pairs and goes quickly from the key to the value. This section covers the map interface, its three basic implementations, and the old `Hashtable` class.

5.5.1 The Map interface and its methods

> **Definition 5.13**
>
> A **map** is a structure that stores key-value pairs. Each key is mapped to one value. Keys are unique; values may repeat.

The best-known analogy for a map is the dictionary. A word (the key) leads to its meaning (the value), and the same word does not appear twice in a dictionary. Two different words, however, can have the same meaning. A map provides fast access by key and makes it possible to operate on pairs. Figure 5.19 shows the hierarchy of the map classes.

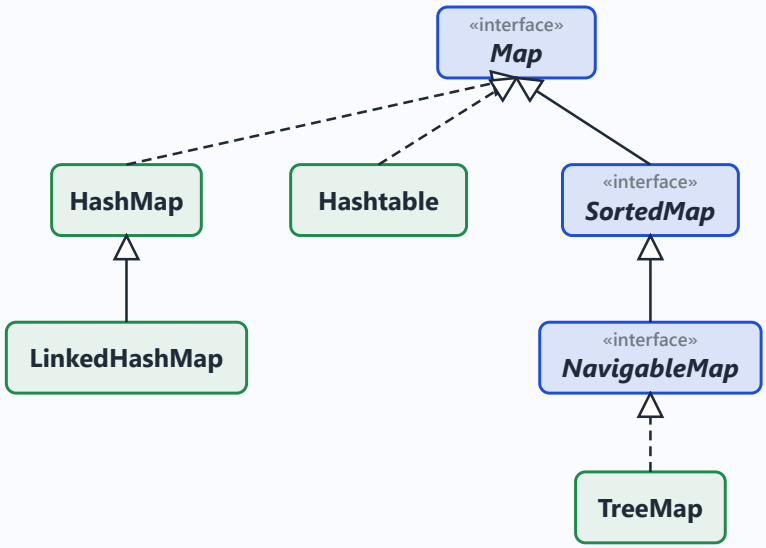


Figure 5.19: The `Map` hierarchy. `HashMap` and `Hashtable` implement `Map`; `LinkedHashMap` extends `HashMap`. `TreeMap` implements `NavigableMap`, which extends `SortedMap`.

The three basic implementations are the map counterpart of the trio among the sets. `HashMap` gives no ordering guarantee. `LinkedHashMap` preserves insertion order. `TreeMap` keeps the keys sorted; the order is the natural order or a given `Comparator`. Table 5.20 gives the methods of the interface.

`K` denotes the key type and `V` the value type. For example, `Map<String, Integer>` is a map from names to grades.

Method	Purpose
`put(K key, V value)`	maps the key to the value; if the key exists, replaces its value and returns the old one
`get(Object key)`	returns the value of the key; `null` if the key is absent
`remove(Object key)`	removes the key and its value
`containsKey(Object key)`	is the key in the map?
`containsValue(Object v)`	is the value in the map?
`keySet()`	returns all keys as a `Set`
`values()`	returns all values as a `Collection`
`entrySet()`	returns all key-value pairs as a `Set<Map.Entry<K, V>>`
`size()`, `isEmpty()`	number of pairs, is it empty?
`getOrDefault(key, v)`	returns `v` if the key is absent (Java 8+)

Table 5.20: The basic methods of the `Map<K, V>` interface.

5.5.2 HashMap

`HashMap` stores key-value pairs in a hash table. It is derived from `AbstractMap` and implements `Map`. The bucket of a pair is computed from the hash code of the key; the hash code of the value is never used. That is why `get` and `put` are O(1) on average. The entries are stored without order.

```
HashMap<String, Double> prices = new HashMap<>();
prices.put("Tablet", 2000.0);
prices.put("Headphones", 350.0);
prices.put("Case", 120.0);
Double old = prices.put("Tablet", 1800.0);
System.out.println(old);
System.out.println(prices.get("Tablet"));
System.out.println(prices.get("Laptop"));
System.out.println(prices.containsKey("Case"));
prices.remove("Case");
System.out.println(prices.size());
```

```
prices.put(null, 0.0);         // a single null key
prices.put("Pen", null);     // null value
System.out.println(prices.size());
```

```
2000.0
1800.0
null
true
2
4
null true 0.0
```

The program proceeds as follows. Three products are added. The fourth call, `put("Tablet", 1800.0)`, does not add a new pair; it changes the value of the existing key. This happens because keys are unique. `put` returns the old value: 2000.0. This return value is also a way to find out whether a key existed before; for a new key, `put` returns `null`. `get("Tablet")` gives the new value. `get("Laptop")` returns `null` for a key that is not present; it does not throw an exception. `containsKey("Case")` returns `true`. After Case is removed, two pairs remain in the map.

`HashMap` allows one `null` key and any number of `null` values. The extra lines add one `null` key and one `null` value; the size becomes 4. The last line, added in this note's program, shows the ambiguity this permission creates. `get("Pen")` returns `null`, but this does not mean "the key is absent"; `containsKey("Pen")` is `true`. In a map that stores `null` values, "absent" and "value is null" can be told apart only with `containsKey`. `getOrDefault("Laptop", 0.0)`, on the other hand, returns the given default when the key is absent: 0.0.

Common mistake

Assigning the result of `get` directly to a primitive type. `int n = counter.get(k);` tries to unbox `null` if the key is absent and throws a `NullPointerException` (its message was given in the section on generic types). Where the key may be missing, `getOrDefault(k, 0)` is used, or `containsKey` is asked first.

5.5.3 Traversing a map

`Map` is not `Iterable`; it cannot be given directly to a for-each loop. Traversal is done through one of the three views a map offers:

```
Map<String, Integer> grades = new TreeMap<>();
grades.put("Cem", 72);
grades.put("Ayse", 95);
grades.put("Banu", 60);

for (String name : grades.keySet()) {
    System.out.print(name + " ");
}
System.out.println();
for (Integer g : grades.values()) {
    System.out.print(g + " ");
}
System.out.println();
for (Map.Entry<String, Integer> g : grades.entrySet()) {
    System.out.println(g.getKey() + " → " + g.getValue());
}
```

```
Ayse Banu Cem
95 60 72
Ayse → 95
Banu → 60
Cem → 72
```

Here the object is a `TreeMap`; the keys come in alphabetical order. `keySet()` gives only the keys: Ayse Banu Cem. `values()` gives only the values, in the order of the keys: 95 60 72. `entrySet()` gives each pair as a `Map.Entry` object. `getKey()` reads the key and `getValue()` the value.

If both key and value are needed, the most efficient way is `entrySet()`. Traversing `keySet()` and calling `get` for each key gives the same result, but looks up the key again at every step. In a `HashMap` this extra lookup is O(1) on average; in a `TreeMap`, each `get` is O(log n) and the total work grows to O(n log n). `entrySet`, on the other hand, already holds the pairs, and the traversal is O(n).

Giving the map itself to for-each is a compile error:

```
MapForEach.java:5: error: for-each not applicable to expression type
        for (String s : m) {
                        ^
  required: array or java.lang.Iterable
  found:    Map<String,Integer>
1 error
```

The message says that for-each can be used only on arrays and `java.lang.Iterable` types, but a `Map<String,Integer>` was found.

5.5.4 Counting words

The classic use of a map is counting the words in a text. Each word is a key, and the value is how many times that word occurs:

```
String text = "egg coffee egg milk coffee egg";
Map<String, Integer> counter = new TreeMap<>();
for (String k : text.split(" ")) {
    if (counter.containsKey(k)) {
        counter.put(k, counter.get(k) + 1);
    } else {
        counter.put(k, 1);
    }
}
System.out.println(counter);
```

```
{coffee=2, egg=3, milk=1}
```

`text.split(" ")` splits the text at spaces and gives an array of six words. The loop handles one of two cases for each word. If the word has been seen before, the old count is taken, increased by one, and written back. If it is seen for the first time, it is added with the value 1. Table 5.21 gives the state after each word.

Step	Line	counter	Variables	New output
1	1: `String text = "egg coffee egg milk coffee egg";` 2: `Map<String, Integer> counter = new TreeMap<>();`	empty	—	—
2	3: `for (String k : text.split(" ")) {` 4: `if (counter.containsKey(k)) {` 7: `counter.put(k, 1);`	egg=1	k="egg"	—
3	3: `for (String k : text.split(" ")) {` 4: `if (counter.containsKey(k)) {` 7: `counter.put(k, 1);`	coffee=1, egg=1	k="coffee"	—
4	3: `for (String k : text.split(" ")) {` 4: `if (counter.containsKey(k)) {` 5: `counter.put(k, counter.get(k) + 1);`	coffee=1, egg=2	k="egg"	—
5	3: `for (String k : text.split(" ")) {` 4: `if (counter.containsKey(k)) {` 7: `counter.put(k, 1);`	coffee=1, egg=2, milk=1	k="milk"	—
6	3: `for (String k : text.split(" ")) {` 4: `if (counter.containsKey(k)) {` 5: `counter.put(k, counter.get(k) + 1);`	coffee=2, egg=2, milk=1	k="coffee"	—

Step	Line	counter	Variables	New output
7	3: `for (String k : text.split(" ")) {` 4: `if (counter.containsKey(k)) {` 5: `counter.put(k, counter.get(k) + 1);`	coffee=2, egg=3, milk=1	k="egg"	—
8	10: `System.out.println(counter);`	coffee=2, egg=3, milk=1	k="egg"	`{coffee=2, egg=3, milk=1}`

Table 5.21: Tracing the word-counting program. The `counter` column is the map at the end of the step.

"egg" and "coffee" are added with 1 when first seen. The third word is "egg" again. `containsKey` returns true; the old value 1 is taken and 2 is written. "milk" is added with 1. "coffee" rises to 2, and the last "egg" to 3. When the loop ends, the `TreeMap` prints the keys in alphabetical order. Figure 5.20 shows three moments of the map.

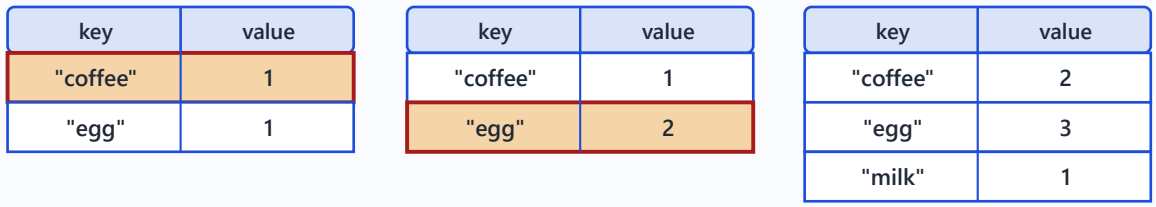


Figure 5.20: The word-counting map. On the left, after two words: coffee=1, egg=1. In the middle, after the second "egg": egg=2. On the right, at the end of the loop: coffee=2, egg=3, milk=1.

The loop body performs at most three map operations per word: `containsKey`, `get`, and `put`. With `getOrDefault` the same job shrinks to one line: `counter.put(k, counter.getOrDefault(k, 0) + 1)`. The `merge` method, introduced in Java 8, is even shorter: `counter.merge(k, 1, Integer::sum)`. If the key is absent this call writes 1; if present, it adds 1 to the old value. The three forms give the same result:

```
{coffee=2, egg=3, milk=1} {coffee=2, egg=3, milk=1}
{egg=3, milk=1, coffee=2}
```

The second line of the output is the same counts copied into a `HashMap`. The order is no longer alphabetical: egg, milk, coffee. The numbers are the same; only the traversal order has changed. If alphabetical output is wanted in word counting, `TreeMap` is chosen; if the fastest counting is wanted, `HashMap`. For a text of `n` words, counting takes O(n) on average with `HashMap` and O(n log n) with `TreeMap`.

5.5.5 LinkedHashMap and Hashtable

LinkedHashMap

`LinkedHashMap` is a map derived from `HashMap` that additionally links its pairs together with a linked list. Thanks to this linked list, insertion order is preserved. It is preferred when speed and order matter together; if order does not matter, `HashMap` is enough. Its basic methods are the same as in `Map`: `put`, `remove`, `get`, `containsKey`, `size`. The difference in order is seen when the same phone book is put into three maps:

```
Map<String, String> contacts = new LinkedHashMap<>();
contacts.put("Zeynep", "555-1111");
contacts.put("Ahmet", "555-1234");
contacts.put("Mert", "555-2222");
Map<String, String> h = new HashMap<>(contacts);
Map<String, String> t = new TreeMap<>(contacts);
System.out.println("LinkedHashMap: " + contacts);
System.out.println("HashMap:       " + h);
System.out.println("TreeMap:       " + t);
```

```
LinkedHashMap: {Zeynep=555-1111, Ahmet=555-1234, Mert=555-2222}
HashMap:       {Ahmet=555-1234, Mert=555-2222, Zeynep=555-1111}
TreeMap:       {Ahmet=555-1234, Mert=555-2222, Zeynep=555-1111}
```

The expressions `new HashMap<>(contacts)` and `new TreeMap<>(contacts)` are constructors that copy all the pairs of the given map. `LinkedHashMap` preserves insertion order: Zeynep, Ahmet, Mert. `TreeMap` arranges the keys in alphabetical order. The order of `HashMap` is the bucket order, and the alphabetical appearance here is a coincidence. The copying constructor chooses the number of buckets according to the size of the source map: 4 buckets for three pairs. With four buckets, the bucket number is the last 2 bits of the hash value. From the hash values of the measurement program in the section on sets, Ahmet falls in bucket 1, Mert in 2, and Zeynep in 3. If the same three names were put into a table with 16 buckets, their buckets would be 1, 14, and 7, and the order would turn into Ahmet, Zeynep, Mert. That the table has 4 buckets was verified with a measurement program that reads the internal array through reflection. The conclusion is clear: the order of a `HashMap` depends on the number of buckets, and programs never rely on this order.

Hashtable

`Hashtable` is an old (Java 1.0), synchronized map class that stores key-value pairs. Its difference from `HashMap` is that each of its methods is locked. Because of the locking, it is slower than `HashMap` in single-threaded programs. In multi-threaded programs, `ConcurrentHashMap` from the `java.util.concurrent` package is a better choice; it uses finer-grained locking instead of locking the whole table. Today `Hashtable` is mostly seen in old code.

The second difference of `Hashtable` is that it does not accept `null`: neither keys nor values can be `null`.

```
Hashtable<String, String> phoneBook = new Hashtable<>();
phoneBook.put("Ahmet", "555-1234");
System.out.println(phoneBook.get("Ahmet"));
System.out.println(phoneBook.containsKey("Ayse"));
// phoneBook.put(null, "x");  → NullPointerException
```

```
555-1234
false
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"Object.hashCode()" because "key" is null
```

In this note's program, the commented line from the slides has been uncommented. The first two outputs are normal; there is no key "Ayse". `put(null, "x")`, however, throws a `NullPointerException`. The message tells the cause of the error: to find the bucket, the table wants to call the key's `hashCode()` method, but the key is `null`. A `null` value is also rejected; `phoneBook.put("Ayse", null)` throws the same exception, this time without a message.

5.5.6 SortedMap and TreeMap

`SortedMap` is the map interface that stores its keys in sorted order; each key appears once. It offers range operations such as the submap and the upper map. `TreeMap` is derived from `AbstractMap` and implements the `NavigableMap` interface. The keys are arranged by the natural order or by a given `Comparator`. Internally there is a red-black tree as in `TreeSet`; `get`, `put`, and `remove` are O(log n). Table 5.22 gives the sorted-map methods. Their names and boundary rules are the same as those of the `TreeSet` methods.

Method	Purpose
`firstKey()` / `lastKey()`	the smallest / the largest key
`headMap(to)`	keys smaller than `to` (excluded)
`tailMap(from)`	`from` and the larger keys (included)
`subMap(from, to)`	`from` included, `to` excluded
`comparator()`	the `Comparator` that orders the map (`null` for the natural order)
`put`, `get`, `remove`	same as `Map`

Table 5.22: The sorted-map methods of `TreeMap`.

```
TreeMap<String, Double> prices = new TreeMap<>();
prices.put("Tablet", 2000.0);
prices.put("Pen", 15.0);
prices.put("Notebook", 40.0);
prices.put("Ruler", 5.0);
System.out.println(prices);
System.out.println(prices.firstKey());
System.out.println(prices.lastKey());
System.out.println(prices.headMap("Pen"));
System.out.println(prices.tailMap("Pen"));
System.out.println(prices.subMap("Notebook", "Ruler"));
```

```
{Notebook=40.0, Pen=15.0, Ruler=5.0, Tablet=2000.0}
Notebook
Tablet
{Notebook=40.0}
{Pen=15.0, Ruler=5.0, Tablet=2000.0}
{Notebook=40.0, Pen=15.0}
```

Whatever order the products are added in, the map is printed in the alphabetical order of the keys: Notebook, Pen, Ruler, Tablet. The smallest key is Notebook and the largest is Tablet. `headMap("Pen")` gives the keys smaller than Pen and does not include Pen: only Notebook. `tailMap("Pen")` includes Pen and the keys larger than it. `subMap("Notebook", "Ruler")` includes Notebook and does not include Ruler. Thanks to the sorted keys, range queries start in O(log n) time; in a `HashMap`, the same query would require scanning all the keys.

5.5.7 Comparing the map classes

	`HashMap`	`LinkedHashMap`	`TreeMap`	`Hashtable`
Traversal order	no guarantee	insertion order	sorted by key	no guarantee
`get` / `put`	O(1) average	O(1) average	O(log n)	O(1) average, locked
`null` key	one	one	no	no
`null` value	yes	yes	yes	no
Synchronized	no	no	no	yes (legacy)
When?	the default choice	order must be preserved	sorted keys, range queries	legacy code

Table 5.23: A comparison of the four map classes.

The logic in the table also holds for sets. `HashSet`, `LinkedHashSet`, and `TreeSet` internally use `HashMap`, `LinkedHashMap`, and `TreeMap`, respectively; the elements of the set are the keys of the map. So the order, speed, and `null` rule of a set are the same as those of the corresponding map.

5.6 The Iterator

A list works with indices, a set with a hash table, a `TreeSet` with a tree, and an `ArrayDeque` with a circular array. Traversing the elements of such different structures with a single form of loop is possible only through a common intermediary. This intermediary is the iterator. Each collection produces an iterator that knows its own internal structure; the user only asks "is there a next element?" and "give me the next one".

5.6.1 The Iterator interface

Definition 5.14

An **iterator** is an object that gives the elements of a collection one by one. It makes it possible to reach every element without knowing the internal structure of the collection. The safe way to remove while traversing is the iterator's `remove()` (for bulk removal, `removeIf`). The `iterator()` method of every `Iterable` collection returns an iterator.

Method	Purpose
`hasNext()`	is there a next element?
`next()`	returns the next element and advances the cursor
`remove()`	removes from the collection the element obtained by the last `next()`

Table 5.24: The methods of the `Iterator<E>` interface.

The easiest way to understand an iterator is to imagine a **cursor**. The cursor sits not on the elements but between them. At the start it is in front of the first element. `next()` makes the cursor hop over one element and returns the element it hopped over. `hasNext()` tells whether any elements remain in front of the cursor. Figure 5.21 shows this arrangement.



Figure 5.21: The iterator's cursor sits between the elements. `next()` has returned 3 and moved the cursor between 3 and 8; `hasNext()` is `true`.

```
ArrayList<Integer> list = new ArrayList<>();
list.add(3);
list.add(8);
list.add(5);
Iterator<Integer> iterator = list.iterator();
while (iterator.hasNext()) {
    Integer num = iterator.next();
    // process the element here
    System.out.println(num);
}
```

```
3
8
5
```

An iterator is usually used with a `while (it.hasNext())` loop. Table 5.25 traces the loop step by step.

Step	Line	List and cursor	Variables	New output
1	1: `ArrayList<Integer> list = new ArrayList<>();` 2: `list.add(3);` 3: `list.add(8);` 4: `list.add(5);`	list 3, 8, 5	—	—
2	5: `Iterator<Integer> iterator = list.iterator();`	list 3, 8, 5; cursor before element 0	—	—
3	6: `while (iterator.hasNext()) {` 7: `Integer num = iterator.next();`	list 3, 8, 5; cursor before element 1	num=3	—
4	9: `System.out.println(num);`	list 3, 8, 5; cursor before element 1	num=3	`3`
5	6: `while (iterator.hasNext()) {` 7: `Integer num = iterator.next();`	list 3, 8, 5; cursor before element 2	num=8	—
6	9: `System.out.println(num);`	list 3, 8, 5; cursor before element 2	num=8	`8`
7	6: `while (iterator.hasNext()) {` 7: `Integer num = iterator.next();`	list 3, 8, 5; cursor at the end	num=5	—
8	9: `System.out.println(num);`	list 3, 8, 5; cursor at the end	num=5	`5`
9	6: `while (iterator.hasNext()) {`	list 3, 8, 5; cursor at the end	num=5	—

Table 5.25: Tracing a traversal with an iterator. The cursor position is given by the index of the element in front of the cursor; index 3 means the end of the list.

`list.iterator()` puts the cursor in front of the first element. In each round of the loop, `hasNext()` returns `true`, `next()` gives the next element and advances the cursor. After three rounds the cursor is at the end. `hasNext()` returns `false` and the loop ends. If `next()` is called when no elements remain, a `NoSuchElementException` is thrown.

The for-each loop is translated behind the scenes into exactly this iterator loop. When you write `for (Integer num : list) { … }`, the compiler creates an iterator and inserts the `hasNext` / `next` calls itself. The Java Language Specification defines this translation precisely. This is also why for-each works only on `Iterable` types.

5.6.2 Removing while traversing

Removing from a list the elements that satisfy a certain condition is a common job. The most natural-looking way is to call the list's own `remove` method inside a for-each. This way is wrong:

```
List<Integer> numbers = new ArrayList<>(Arrays.asList(4, 7, 10, 13, 16));
for (Integer s : numbers) {
    if (s % 2 == 0) {
        numbers.remove(s);
    }
}
```

```
Exception in thread "main" java.util.ConcurrentModificationException
```

The first even number, 4, is removed, but right after that the program stops with a `ConcurrentModificationException`. The correct way is to write the iterator explicitly and call its own `remove()` method:

```
List<Integer> numbers = new ArrayList<>();
Collections.addAll(numbers, 4, 7, 10, 13, 16);
Iterator<Integer> it = numbers.iterator();
while (it.hasNext()) {
    if (it.next() % 2 == 0) {
        it.remove();         // safe removal
    }
}
System.out.println(numbers);
```

`it.next()` takes the element, and it is checked whether it is even. If it is, `it.remove()` removes the element last taken from the list. Since the iterator does the removal itself, it also updates its internal state and the traversal continues undisturbed. Java 8 brought a short way to do the same job: `removeIf`. The condition is given as a lambda expression and the method builds the loop itself:

```
List<Integer> s2 = new ArrayList<>();
Collections.addAll(s2, 4, 7, 10, 13, 16);
s2.removeIf(s -> s % 2 == 0);
System.out.println(s2);
```

```
[7, 13]
[7, 13]
```

The output of both ways is the same: only the odd numbers remain. `ArrayList.removeIf` first marks the elements to be removed, then shifts the remaining ones left in a single pass. So the total work is O(n). Removing one by one with an iterator, on the other hand, shifts the elements behind on each removal and in the worst case grows to $O(n^2)$. On large lists `removeIf` is both shorter and faster.

`it.remove()` can be called at most once after each `next()`; otherwise an `IllegalStateException` is thrown.

Common mistake

Removing while moving forward in an indexed loop. The loop `for (int i = 0; i < a.size(); i++) if (a.get(i) % 2 == 0) a.remove(i);` does not throw an exception, but can give a wrong result. When an element is removed, the element behind it shifts into its index and `i++` skips that element. In the list `[4, 10, 7, 13, 16]`, 4 is removed, 10 shifts to index 0 and is never checked. In this note's program the result was `[10, 7, 13]`. If the same loop is built from the end toward the start, the shifting does not affect unchecked elements and the result comes out right: `[7, 13]`.

5.6.3 Where does ConcurrentModificationException come from?

To understand why removal inside a for-each throws an exception, we need to look inside the iterator. `ArrayList` has a field `modCount`. This field is the list's structural-modification counter: every insertion and removal increases it by one. At the moment the iterator is created, it stores the value of this counter as `expectedModCount` and also keeps a `cursor`. Each `next()` call first compares the two counters. If they differ, the list has been modified without the iterator's knowledge and it is not safe to continue the traversal. In this case the iterator throws a `ConcurrentModificationException`. This behavior is called *fail-fast*. The fail-fast check is best effort, not a guarantee: some modifications can pass without an exception, so the correctness of a program is never based on this exception.

```
List<Integer> s = new ArrayList<>();
Collections.addAll(s, 6, 9, 12);
for (Integer x : s) {      // hidden Iterator
    if (x % 2 == 0) {
        s.remove(x);
    }
}
System.out.println(s);
```

Table 5.26 traces the program together with the counters.

Step	Line	Variables	New output
1	1: `List<Integer> s = new ArrayList<>();` 2: `Collections.addAll(s, 6, 9, 12);`	s=[6, 9, 12], modCount=0	—
2	3: `for (Integer x : s) { // hidden Iterator`	s=[6, 9, 12], modCount=0	—
3	3: `for (Integer x : s) { // hidden Iterator`	s=[6, 9, 12], modCount=0, hasNext=true	—
4	3: `for (Integer x : s) { // hidden Iterator`	s=[6, 9, 12], modCount=0, x=6	—
5	4: `if (x % 2 == 0) {`	s=[6, 9, 12], modCount=0, x=6	—
6	5: `s.remove(x);`	s=[9, 12], modCount=1, x=6	—
7	3: `for (Integer x : s) { // hidden Iterator`	s=[9, 12], modCount=1, hasNext=true	—
8	3: `for (Integer x : s) { // hidden Iterator`	s=[9, 12], modCount=1	`Exception in thread "main" java.util.ConcurrentModificationException`

Table 5.26: Tracing removal inside a for-each.

The list has three elements and `modCount` is 0. The for-each secretly calls `s.iterator()`: `cursor = 0`, `expectedModCount = 0`. The first `hasNext()` returns `true` because `cursor` (0) and `size` (3) differ. `next()` first compares the counters (0 = 0), returns 6, and sets `cursor` to 1. 6 is even and `s.remove(x)` is called. This is the list's own method: 9 and 12 shift left, `size` drops to 2, and `modCount` rises to 1. The iterator knows nothing about this. Figure 5.22 shows this moment and what follows.

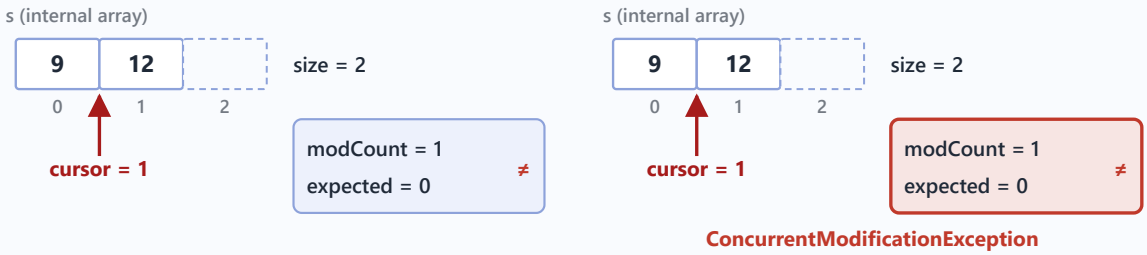


Figure 5.22: On the left, after `s.remove(6)` : the list is `[9, 12]` , `modCount = 1` , the iterator's `expected` value is still 0, and `cursor = 1` . On the right, the following `next()` : the counters do not match and a `ConcurrentModificationException` is thrown.

The next `hasNext()` returns `true` because `cursor` (1) and `size` (2) differ; the loop continues. But when `next()` compares the counters, `modCount` is 1 and `expectedModCount` is 0. The exception is thrown and `println` is never reached.

```
Exception in thread "main" java.util.ConcurrentModificationException
```

The trace shows one more thing: the exception appears not at the moment of removal but at the next `next()` . If `hasNext()` returns `false` after the removal, `next()` is never called and no exception occurs. Had the list been `{6, 9}` , after 6 is removed `cursor` would be 1 and `size` 1. `hasNext()` would return `false` and the loop would end silently:

```
[9]
```

The program ends without error, but 9 was never checked. Wrong code sometimes seems to work. That is why saying "it worked for me" is not enough. Removal while traversing is always done with the iterator's `remove` or with `removeIf` . The word "concurrent" in the name of the exception can be misleading. This error is not specific to multi-threaded programs; it occurs in a single-threaded program too, as it does here.

5.7 Algorithms

The third element of the Collections Framework is the ready-made algorithms. Most of them are gathered in the `java.util.Collections` class. This section covers, in turn, the algorithms for sorting, defining an order (`Comparable` and `Comparator`), shuffling, searching, and composition.

5.7.1 The Collections class

Two names are often confused. `Collection` is an interface and defines the methods common to all collections. `Collections` (with an "s" at the end) is a utility class that gathers ready-made `static` methods that operate on collections. No objects are created from this class; its methods are called with the class name, as in `Collections.sort(list)`.

These algorithms are fast and tested; writing the same job by hand both takes time and carries the risk of errors. They fall into four groups:

- **Sorting**: `sort`, `reverse`, `reverseOrder`.
- **Shuffling**: `shuffle`.
- **Searching**: `binarySearch`, `max`, `min`, `frequency`.
- **Composition**: `copy`, `fill`, `swap`, `reverse`.

5.7.2 Sorting

```
List<String> names = new ArrayList<>(
        Arrays.asList("Cem", "Ayse", "Banu", "Ali"));
Collections.sort(names);
System.out.println(names);
Collections.sort(names, Collections.reverseOrder());
System.out.println(names);
Collections.reverse(names);
System.out.println(names);
```

```
[Ali, Ayse, Banu, Cem]
[Cem, Banu, Ayse, Ali]
[Ali, Ayse, Banu, Cem]
```

`Collections.sort(names)` sorts the list by its natural order, that is, alphabetically. The second call takes a comparator: `Collections.reverseOrder()` reverses the natural order and the list is arranged from Z to A. `Collections.reverse`, on the other hand, makes no comparisons at all; it reverses the current order of the list. When a list in descending order is reversed, ascending order is obtained again. Table 5.27 summarizes the three methods.

Method	Purpose
`sort(list)`	by the natural order; the elements must be `Comparable`
`sort(list, c)`	by the given comparator (`Comparator`)
`reverse(list)`	reverses the list

Table 5.27: Sorting methods.

Classes such as `String`, `Integer`, and `Double` are already `Comparable`. Since Java 8 the list also has its own `sort` method: `list.sort(null)` sorts by the natural order and `list.sort(c)` by a comparator. If a list whose elements cannot be compared is given to `Collections.sort`, the error is caught at compile time:

```
StudentSort.java:9: error: no suitable method found for sort(List<Student>)
        Collections.sort(l);
                   ^
    method Collections.<T#1>sort(List<T#1>) is not applicable
      (inference variable T#1 has incompatible bounds
        equality constraints: Student
        upper bounds: Comparable<? super T#1>)
    method Collections.<T#2>sort(List<T#2>,Comparator<? super T#2>) is not applicable
      (cannot infer type-variable(s) T#2
        (actual and formal argument lists differ in length))
  where T#1,T#2 are type-variables:
    T#1 extends Comparable<? super T#1> declared in method <T#1>sort(List<T#1>)
    T#2 extends Object declared in method <T#2>sort(List<T#2>,Comparator<? super T#2>)
1 error
```

The message is long, but its essence is on a single line: the type `Student` does not satisfy the upper bound `Comparable<? super T>`; that is, it is not comparable. The second form suggested, `sort(List, Comparator)`, does not fit either because the number of arguments does not match. Unlike the run-time error in the priority queue, here the compiler sees the problem in advance; the signature of `sort` restricts the type parameter.

`Collections.sort` uses the TimSort algorithm on object lists. TimSort is an adapted variant of merge sort. It takes O(n log n) in the worst case and is much faster on partially sorted data. It is also **stable**: the relative order of elements considered equal is preserved. `Arrays.sort(int[])`, which sorts primitive arrays, uses Dual-Pivot Quicksort; this choice was made because stability is meaningless for primitive values. This note's program sorted the list "Cem:B, Ali:A, Ece:B, Banu:A, Can:A" by letter grade only. The result is `[Ali:A, Banu:A, Can:A, Cem:B, Ece:B]`: those with the same grade keep their old order in the list. In this way, when an alphabetical list is sorted by grade, those with the same grade stay alphabetical.

5.7.3 Comparable: natural ordering

Definition 5.15

Natural ordering is the default order that a class defines by implementing the `Comparable<T>` interface and writing the `compareTo` method. If `a.compareTo(b)` is negative, `a` comes first; if 0, the two are equal; if positive, `b` comes first.

```
class Student implements Comparable<Student> {
    String name;
    int score;

    Student(String name, int score) {
        this.name = name;
        this.score = score;
    }

    // ascending by score
    @Override
    public int compareTo(Student other) {
        return Integer.compare(score, other.score);
    }

    @Override
    public String toString() {
        return name + ":" + score;
    }
}
```

```
List<Student> roster = new ArrayList<>();
roster.add(new Student("Cem", 72));
roster.add(new Student("Ayse", 95));
roster.add(new Student("Banu", 60));

Collections.sort(roster);
System.out.println(roster);
```

```
[Banu:60, Cem:72, Ayse:95]
[Ayse:95, Banu:60, Cem:72]
[Banu:60, Cem:72, Ayse:95]
[Ayse:95, Cem:72, Banu:60]
```

The `Student` class implements `Comparable<Student>` and defines ascending order by score. `Collections.sort(roster)` uses this order: Banu (60), Cem (72), Ayse (95). The following lines of the program are comparator examples and are explained in the next subsection.

In the body of `compareTo`, `Integer.compare(score, other.score)` is used. It looks as though the same job could be done with `return score - other.score;`: if the difference is negative, `this` is smaller. As long as the scores are small numbers, this works correctly. But the difference of two very large numbers of opposite sign exceeds the `int` range and the sign flips:

```
public class Overflow {
    public static void main(String[] args) {
        int a = 2_000_000_000, b = -2_000_000_000;
        System.out.println((a - b) + " " + Integer.compare(a, b));
    }
}
```

```
-294967296 1
```

2 billion is greater than −2 billion, and the difference should be positive. But 4 billion does not fit in an `int`, and the result comes out negative; the sorting would be wrong. `Integer.compare` does not subtract; it only compares, and gives the correct result (1). So instead of subtraction, always use `Integer.compare` (or `Double.compare`).

There are a few rules a good `compareTo` must obey. If `a` comes before `b`, then `b` must come after `a`. If `a` comes before `b` and `b` before `c`, then `a` must come before `c`. It is also recommended that objects for which `compareTo` returns 0 have `equals` return true. This last rule matters for `TreeSet` and `TreeMap`: these structures test equality not with `equals` but with `compareTo`. If two different students with the same score are put into a `TreeSet`, the second is considered a "duplicate" and is not added.

5.7.4 Comparator: another order

Definition 5.16

A **comparator** is a separate ordering criterion written outside the class. It implements the `Comparator<T>` interface and compares two objects with the `compare(a, b)` method. A class has a single natural order but can have as many comparators as desired.

```
Collections.sort(roster, new Comparator<Student>() {
    @Override
    public int compare(Student a, Student b) {
        return a.name.compareTo(b.name);   // by name
    }
});
System.out.println(roster);

// Java 8+: the same, shorter
roster.sort(Comparator.comparing(o -> o.score));
System.out.println(roster);
```

This code is the continuation of the previous program and produces the second and third lines of the output. In the classic form, the comparator is written as an **anonymous class**: `new Comparator<Student>() { … }` is a class that has no name, is defined in place, and has only one object created. This comparator compares the names with the `compareTo` of `String`, and the list becomes alphabetical: Ayse, Banu, Cem.

With Java 8 the same job shrinks to one line. `Comparator.comparing(o -> o.score)` takes a function that extracts the score from each student and produces a comparator that compares by these values. The list returns to score order. For reverse order, `.reversed()` is added. The last line in this note's program does this: with `Comparator.comparing((Student o) -> o.score).reversed()`, the list starts with the highest score. In this form the parameter type of the lambda must be given explicitly; the `.reversed()` chain prevents the compiler from inferring the type from the left.

5.7.5 Shuffling

Shuffling puts the elements of a collection into a random order. Dealing cards, drawing lots, and randomizing the order of questions in an exam are typical uses. `Collections.shuffle(list)` uses the Fisher–Yates (Knuth) algorithm. The algorithm starts from the end of the list. At each step, it picks one at random from among the elements not yet placed and swaps it with the element at the current position. In a list of `n` elements, `n - 1` swaps are made and the work is O(n). The valuable property of the algorithm is that every possible ordering occurs with equal probability (uniform distribution).

```
List<String> students = new ArrayList<>(
        Arrays.asList("Ali", "Banu", "Cem", "Deniz"));
Collections.shuffle(students);
System.out.println(students);
```

The output of this program differs on every run; the `[Cem, Ali, Deniz, Banu]` in the slides is only one of the 24 possible orderings. When writing tests, or when a result must be reproduced, the same shuffle is wanted. For this, a random number generator with a fixed seed is given as the second argument: `Collections.shuffle(students, new Random(42))`. This note's program shuffles the same list twice with seed 42, and the two results are exactly the same:

```
[Deniz, Banu, Ali, Cem]
[Deniz, Banu, Ali, Cem]
```

The lottery example

A nice application of shuffling is random selection without repetition. The numbers from 1 to 49 are put in a list, the list is shuffled, and the first 6 elements are taken:

```
public class Lottery {
    // draws lottery numbers
    public static Integer[] draw(int total, int count) {
        ArrayList<Integer> numbers = new ArrayList<Integer>();
        for (int i = 0; i < total; i++) {
            numbers.add(i + 1);          // 1 … total
        }
        Collections.shuffle(numbers);    // shuffle
        Integer[] drawn = new Integer[count];
        for (int i = 0; i < count; i++) {
            drawn[i] = numbers.get(i);
        }
        return drawn;
    }

    public static void main(String[] args) {
        Integer[] result = draw(49, 6);   // 6 out of 49
        for (Integer item : result) {
            System.out.print(item + " ");
        }
    }
}
```

The output of one run is as follows (it differs on every run):

```
21 25 16 3 33 46
```

Because shuffling only moves the numbers around, every number occurs exactly once in the list. So the first six elements are free of repeats, too. This note's test program called `draw(49, 6)` 10,000 times and checked that every result contained 6 distinct numbers in the range 1–49; the number of errors is 0. The cost is the cost of building and shuffling the whole list, independent of the count of numbers selected: `total` numbers, that is, O(n).

5.7.6 Searching

There are two basic ways to search for an element in a collection.

Linear search
: Checks the elements one by one from start to finish. The collection need not be sorted. It is effective on small collections. Its cost is O(n). `indexOf` and `contains` perform linear search.

Binary search
: In a sorted collection, it looks at the middle element. If the sought value is smaller, it continues the search in the left half; if larger, in the right half. At each step the range is halved. Its cost is O(log n). The collection must be sorted.

```
List<Integer> numbers = Arrays.asList(1, 3, 5, 7, 9, 11);
System.out.println(Collections.binarySearch(numbers, 7));
System.out.println(Collections.binarySearch(numbers, 6));
System.out.println(numbers.indexOf(7));    // linear search
```

```
3
-4
3
-1 -7
2
```

The first three lines are the output in the slides. `binarySearch(numbers, 7)` finds 7 at index 3. The search proceeds in these steps. The range is 0–5; the middle index 2 holds 5, which is smaller than 7, and the search moves to the right half. The range is 3–5; the middle index 4 holds 9, which is greater than 7, and it moves to the left half. The range is 3–3 and the 7 at index 3 is found. Three comparisons were made. This note's counting program verifies this number by adding a counter to the comparator:

```
7: result 3, 3 comparisons
6: result -4, 3 comparisons
1 million elements, max over tried values: 20
```

6 is not in the list. When the search ends, the place where 6 should go, that is, the **insertion point**, is 3: after 5 and before 7. For a value that is not found, the return is `-(insertion point) - 1`: −(3) − 1 = −4. This form carries two pieces of information at once. If the result is negative, the element is absent; `-(result) - 1` gives the place where the element would be inserted without breaking the order. The answer to "why not just −3?" is 0: if the insertion point were 0, −0 = 0 would result and be confused with "found at index 0". The fourth line of this note's program shows the edge cases. When 0 is searched for, the insertion point is 0 and the return is −1. When 12 is searched for, the insertion point is 6 (the end of the list) and the return is −7.

If the list is not sorted, the result is undefined. The last line of the program searches for 7 in an unsorted list and returns 2. 7 is indeed at index 2, but this correct result is a coincidence. Had another value been searched for, a wrong index or "absent" could have come back. `binarySearch` does not check whether the list is sorted; this check is not made because it would take O(n).

The difference between the two searches is striking on large data. In a million elements, linear search makes up to a million comparisons. Binary search makes at most 20 comparisons, because $2^{20} \approx 1$ million. The last line of the counting program shows that among the values tried in a list of a million elements, at most 20 comparisons were made. A list that is searched often is sorted once and then used with binary search. Sorting takes O(n log n), but every later search is O(log n).

5.7.7 Composition algorithms

Composition algorithms copy the elements of a list, change them, swap their places, or count them. Table 5.28 collects them.

Method	Purpose
`copy(dest, src)`	copies the elements of the source to the beginning of the destination; `dest.size() ≥ src.size()` is required
`fill(list, d)`	sets all elements to `d`
`reverse(list)`	reverses the order of the elements
`swap(list, i, j)`	swaps positions `i` and `j`
`max` / `min`	the largest / the smallest element
`frequency(c, o)`	how many times `o` occurs

Table 5.28: Composition algorithms.

```
List<Integer> n = Arrays.asList(4, 9, 2, 9, 5);
System.out.println(Collections.max(n));
System.out.println(Collections.min(n));
System.out.println(Collections.frequency(n, 9));
```

```
9
2
2
```

`max` and `min` look at all the elements of the list and give the largest and the smallest by natural order: 9 and 2. `frequency(n, 9)` counts how many times 9 occurs: 2. All three traverse the list once and are O(n). The list need not be sorted.

```
List<String> names =
        Arrays.asList("Ahmet", "Ayse", "Cem");
List<String> target = new ArrayList<>(
        Arrays.asList("X", "Y", "Z", "W"));
Collections.copy(target, names);
System.out.println(target);
Collections.reverse(target);
System.out.println(target);
Collections.swap(target, 0, 2);
System.out.println(target);
Collections.fill(target, "Java");
System.out.println(target);
```

```
[Ahmet, Ayse, Cem, W]
[W, Cem, Ayse, Ahmet]
[Ayse, Cem, W, Ahmet]
[Java, Java, Java, Java]
```

Table 5.29 gives the four operations step by step.

Step	Line	Lists	New output
1	1: `List<String> names =` 2: `Arrays.asList("Ahmet", "Ayse", "Cem");` 3: `List<String> target = new ArrayList<>(` 4: `Arrays.asList("X", "Y", "Z", "W"));`	Ahmet, Ayse, Cem; target: X, Y, Z, W	—
2	5: `Collections.copy(target, names);` 6: `System.out.println(target);`	Ahmet, Ayse, Cem; target: Ahmet, Ayse, Cem, W	`[Ahmet, Ayse, Cem, W]`
3	7: `Collections.reverse(target);` 8: `System.out.println(target);`	Ahmet, Ayse, Cem; target: W, Cem, Ayse, Ahmet	`[W, Cem, Ayse, Ahmet]`
4	9: `Collections.swap(target, 0, 2);` 10: `System.out.println(target);`	Ahmet, Ayse, Cem; target: Ayse, Cem, W, Ahmet	`[Ayse, Cem, W, Ahmet]`
5	11: `Collections.fill(target, "Java");` 12: `System.out.println(target);`	Ahmet, Ayse, Cem; target: Java, Java, Java, Java	`[Java, Java, Java, Java]`

Table 5.29: Tracing the composition operations.

`names` is a fixed-size list created with `Arrays.asList`. `target` is a modifiable `ArrayList` with four elements. `copy` writes the three elements of the source to positions 0, 1, and 2 of the target. The fourth element of the target, W, stays in place; `copy` does not change the length of the target. `reverse` reverses the order: `[W, Cem, Ayse, Ahmet]`. `swap(target, 0, 2)` exchanges positions 0 and 2: W and Ayse change places. `fill` sets all elements to "Java"; the size is still 4. Figure 5.23 shows two intermediate states.



Figure 5.23: On the left, after `copy`: target is `[Ahmet, Ayse, Cem, W]`. On the right, after `reverse` and `swap(target, 0, 2)`: `[Ayse, Cem, W, Ahmet]`.

The destination requirement of `copy` often causes mistakes. If the destination is shorter than the source, an exception is thrown. The real trap is a list that has enough capacity but is empty. `new ArrayList<>(10)` allocates 10 cells, but its size is 0; `copy` looks at the size, not the capacity:

```
import java.util.*;
public class EmptyTarget {
    public static void main(String[] args) {
        List<String> source = Arrays.asList("a", "b", "c");
        List<String> target = new ArrayList<>(10);
        System.out.println(target.size());
        Collections.copy(target, source);
    }
}
```

```
0
Exception in thread "main" java.lang.IndexOutOfBoundsException: Source does not fit in
dest
```

If an independent copy of a list is needed, the copying constructor is used instead of `copy`: `new ArrayList<>(source)`. This way does not care about the size of the target and is a single line.

5.8 The Stream API

Most of the jobs in the earlier sections follow the same pattern: traverse a collection, select some elements, transform them, and collect the result in a new collection. This pattern requires writing a loop, a condition, and an intermediate list every time. The Stream API, introduced in Java 8, makes it possible to write the same job as a pipeline.

5.8.1 Streams and pipelines

Definition 5.17

A **stream** is an object in the `java.util.stream` package that processes a sequence of data as a **pipeline**. The pipeline consists of a source, zero or more intermediate operations, and a terminal operation. The programmer says "what to do"; the library builds the loop.

The stream here is a different concept from input/output (I/O) streams such as `InputStream`; only the names are the same. A stream does not store data. It passes over a source (a list, an array, the lines of a file) and does not modify the source. Figure 5.24 shows a pipeline.

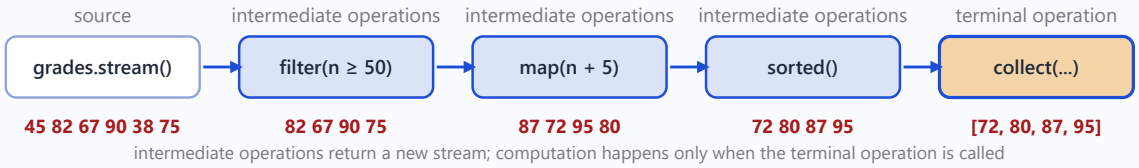


Figure 5.24: A stream pipeline. The source is `grades.stream()`; the intermediate operations are `filter(n ≥ 50)`, `map(n + 5)`, and `sorted()`; the terminal operation is `collect(...)`. Under each box are the values that pass through that stage. Intermediate operations return a new stream; the pipeline runs only when the terminal operation is called.

```
List<Integer> grades = Arrays.asList(45, 82, 67, 90, 38, 75);
List<Integer> passed = grades.stream()
    .filter(n -> n >= 50)
    .map(n -> n + 5)
    .sorted()
    .collect(Collectors.toList());
System.out.println(passed);
```

```
[72, 80, 87, 95]
[45, 82, 67, 90, 38, 75]
[72, 80, 87, 95]
[72, 80, 87, 95]
```

The pipeline has four steps. `filter` lets through the grades of 50 and above: 82, 67, 90, 75. `map` adds 5 to each: 87, 72, 95, 80. `sorted` sorts them. `collect(Collectors.toList())` gathers the result in a new list. In the note's program, the second line shows that the source did not change, the third line shows the short form in Java 16 (`.toList()`), and the fourth line shows the same job written with a loop. The loop version needs an intermediate list, a condition, and a separate sort call. The stream version names each step. The required imports are `java.util.*` and `java.util.stream.Collectors`.

There is a small difference between `Collectors.toList()` and `.toList()`. The first returns a modifiable list (in practice an `ArrayList`). The second returns an unmodifiable list; if `add` is called on it, an `UnsupportedOperationException` is thrown.

5.8.2 Intermediate and terminal operations

Intermediate operation	Purpose	Terminal operation	Result
`filter(p)`	lets through those that satisfy the condition	`collect(...)`	list, set, map...
`map(f)`	transforms each element	`forEach(f)`	an action for each element

Intermediate operation	Purpose	Terminal operation	Result
`flatMap(f)`	flattens nested streams	`count()`	number of elements (`long`)
`sorted()`	sorts (waits for all elements)	`reduce(...)`	reduces the elements to a single value
`distinct()`	discards duplicates (`equals`)	`findFirst()`	the first element (`Optional`)
`limit(n)` / `skip(n)`	takes / skips the first `n` elements	`anyMatch` / `allMatch` / `noneMatch`	`boolean`
`peek(f)`	observes the elements (debugging)	`min` / `max`	`Optional`

Table 5.30: Intermediate and terminal operations.

An intermediate operation returns a new stream and the chain continues. A terminal operation returns a result outside the stream (or nothing) and runs the pipeline. A pipeline without a terminal operation does nothing. A few of the operations in the table were tried in this note's program:

```
import java.util.*;
import java.util.stream.*;
public class Main68 {
    public static void main(String[] args) {
        List<List<Integer>> nested = Arrays.asList(Arrays.asList(1, 2), Arrays.asList(3),
Arrays.asList(4, 5));

System.out.println(nested.stream().flatMap(List::stream).collect(Collectors.toList()));
        System.out.println(Stream.of(3, 1, 3, 2,
1).distinct().collect(Collectors.toList()));
        System.out.println(Stream.of(1, 2, 3, 4, 5,
6).skip(2).limit(3).collect(Collectors.toList()));
        System.out.println(Stream.of(4, 8, 15).count());
        System.out.println(Stream.of(4, 8, 15).anyMatch(x -> x > 10) + " " + Stream.of(4,
8, 15).allMatch(x -> x > 3)
                + " " + Stream.of(4, 8, 15).noneMatch(x -> x > 20));
        System.out.println(Stream.of(4, 8, 15).max(Integer::compare) + " " + Stream.
<Integer>empty().findFirst());
        Stream.of("a", "b").peek(s -> System.out.print("peek " + s + "; ")).forEach(s ->
System.out.print("forEach " + s + "; "));
        System.out.println();
        long n = Stream.of(1, 2, 3).peek(x -> System.out.print("unseen ")).count();
        System.out.println("count " + n);
    }
}
```

```
[1, 2, 3, 4, 5]
[3, 1, 2]
[3, 4, 5]
3
true true true
Optional[15] Optional.empty
peek a; forEach a; peek b; forEach b;
count 3
```

`flatMap` unrolls the three inner lists into a single stream of numbers. `distinct` discards the repeated 3 and 1 and preserves the order of first appearance. `skip(2).limit(3)` skips the first two elements and takes the next three. `max` gives its result inside an `Optional`; on an empty stream `findFirst` returns `Optional.empty`. The `peek` line shows that the elements pass through the pipeline one at a time: "peek a" is immediately followed by "forEach a". The last line is a warning. Because JDK 21 knows the result of `count()` directly from the size of the source, it does not run the pipeline at all and the text in `peek` is never seen. `peek` is for debugging only; the real work is not put inside it.

5.8.3 Lazy evaluation and single use

Definition 5.18

Lazy evaluation: when intermediate operations are called, they process no elements; they merely record a recipe. The elements are processed when the terminal operation is called, passing through the pipeline one at a time.

```
List<String> names =
    Arrays.asList("Arda", "Ece", "Ayse", "Can");
Stream<String> pipe = names.stream()
    .filter(s -> s.startsWith("A"))
    .map(String::toUpperCase);
// nothing processed yet.
List<String> result =
    pipe.collect(Collectors.toList());
System.out.println(result);  // [ARDA, AYSE]
pipe.count();   // IllegalStateException
```

```
[ARDA, AYSE]
Exception in thread "main" java.lang.IllegalStateException: stream has already been
operated upon or closed
```

The `pipe` variable is a `Stream<String>`. When the first three lines have run, no name has yet been filtered and no letter capitalized. When `collect` is called, the pipeline runs and the result is `[ARDA, AYSE]`. `String::toUpperCase` is a **method reference**; it is the short form of the lambda `s -> s.toUpperCase()`. The last line shows the second rule: a stream is **single-use**. If the same stream is used again after a terminal operation, an `IllegalStateException` is thrown. If necessary, a new stream is taken from the source: `names.stream()`.

Laziness has two important consequences. First, the elements pass through the pipeline *one at a time*. The first element goes through `filter` and `map`, and only then does the second element arrive. Second, operations such as `findFirst` and `limit` stop when they find the result and the remaining elements are never processed. This makes it possible to work even with an infinite source: `Stream.iterate(1, x -> x * 2).filter(x -> x > 100).findFirst()` finds 128 in a stream that goes on forever as 1, 2, 4, … and stops. `sorted` is the exception to this rule: to be able to sort, it must first collect all the elements.

```
import java.util.*;
import java.util.stream.*;
public class SortedWaits {
    public static void main(String[] args) {
        List<Integer> l = Stream.of(3, 1, 2)
            .peek(x -> System.out.println("before " + x))
            .sorted()
            .peek(x -> System.out.println("after " + x))
            .collect(Collectors.toList());
        System.out.println(l);
    }
}
```

```
before 3
before 1
before 2
after 1
after 2
after 3
[1, 2, 3]
```

The three "before" lines come before the "after" lines; `sorted` is a waiting point in the pipeline; it passes no element forward until all the elements have arrived.

5.8.4 Tracing a stream pipeline step by step

To see laziness with our own eyes, let us add a print line to each of `filter` and `map`:

```
List<String> names = Arrays.asList("Arda", "Ece", "Ayse", "Can");
Stream<String> pipe = names.stream()
    .filter(s -> { System.out.println("filter " + s);
                   return s.startsWith("A"); })
    .map(s -> { System.out.println("map " + s);
                return s.toUpperCase(); });
System.out.println("pipe ready");
List<String> result = pipe.collect(Collectors.toList());
System.out.println(result);
```

```
pipe ready
filter Arda
map Arda
filter Ece
filter Ayse
map Ayse
filter Can
[ARDA, AYSE]
```

Step	Line	Variables	New output
1	1: `List<String> names = Arrays.asList("Arda", "Ece", "Ayse", "Can");` 2: `Stream<String> pipe = names.stream()` 3: `.filter(s -> { System.out.println("filter " + s);` 4: `return s.startsWith("A"); })` 5: `.map(s -> { System.out.println("map " + s);` 6: `return s.toUpperCase(); });`	names=[Arda, Ece, Ayse, Can], pipe=Stream	—
2	7: `System.out.println("pipe ready");`	pipe=Stream	`pipe ready`
3	8: `List<String> result = pipe.collect(Collectors.toList());` 3: `.filter(s -> { System.out.println("filter " + s);` 4: `return s.startsWith("A"); })`	s="Arda"	`filter Arda`
4	8: `List<String> result = pipe.collect(Collectors.toList());` 5: `.map(s -> { System.out.println("map " + s);` 6: `return s.toUpperCase(); });`	s="Arda"	`map Arda`
5	8: `List<String> result = pipe.collect(Collectors.toList());` 3: `.filter(s -> { System.out.println("filter " + s);` 4: `return s.startsWith("A"); })`	s="Ece"	`filter Ece`
6	8: `List<String> result = pipe.collect(Collectors.toList());` 3: `.filter(s -> { System.out.println("filter " + s);` 4: `return s.startsWith("A"); })`	s="Ayse"	`filter Ayse`

Step	Line	Variables	New output
7	8: `List<String> result = pipe.collect(Collectors.toList());` 5: `.map(s -> { System.out.println("map " + s);` 6: `return s.toUpperCase(); });`	s="Ayse"	`map Ayse`
8	8: `List<String> result = pipe.collect(Collectors.toList());` 3: `.filter(s -> { System.out.println("filter " + s);` 4: `return s.startsWith("A"); })`	s="Can"	`filter Can`
9	8: `List<String> result = pipe.collect(Collectors.toList());`	result=[ARDA, AYSE]	—
10	9: `System.out.println(result);`	result=[ARDA, AYSE]	`[ARDA, AYSE]`

Table 5.31: Tracing the lazy stream pipeline.

While the pipeline is being built, no lambda runs and there is no output. That is why the "pipe ready" line is printed before all the operations. When `collect` runs the pipeline, the first element, Arda, enters `filter`, passes because it starts with "A", and goes straight to `map`. Only then is it the second element's turn. Ece does not satisfy the condition and never reaches `map`. Ayse passes and is capitalized. Can is eliminated. In the end `filter` ran 4 times and `map` only 2 times. Figure 5.25 shows two moments.

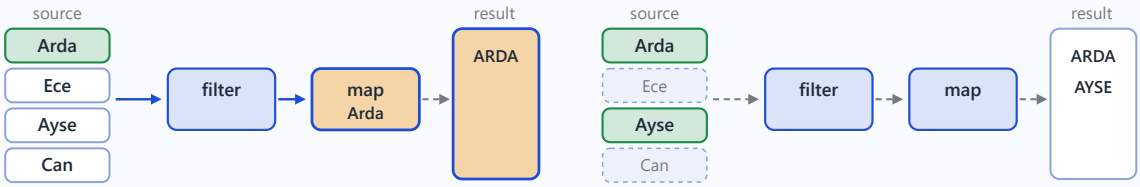


Figure 5.25: On the left, Arda has passed `filter`, reached `map`, and ARDA has been added to the result; the other elements still wait in the source. On the right, the pipeline has finished: the result holds ARDA and AYSE.

If `findFirst` is used instead of `collect` on the same pipeline, the pipeline stops as soon as it finds the first result:

```
filter Arda
map Arda
ARDA
```

Only Arda has been processed; no lambda was called for Ece, Ayse, or Can. This is the counterpart of `break` in code written with a loop.

5.8.5 reduce, Collectors, and primitive streams

```
List<String> fruits = Arrays.asList(
    "pear", "kiwi", "fig", "plum", "lemon");
int total = Arrays.asList(4, 8, 15, 16, 23)
    .stream().reduce(0, Integer::sum);
Map<Integer, List<String>> byLen = fruits
    .stream().collect(
        Collectors.groupingBy(String::length));
String all = fruits.stream()
    .collect(Collectors.joining(", "));
int t = IntStream.rangeClosed(1, 100).sum();
Optional<String> z = fruits.stream()
    .filter(m -> m.startsWith("z")).findFirst();
System.out.println(z.orElse("none"));
```

```
none
total = 66
byLen = {3=[fig], 4=[pear, kiwi, plum], 5=[lemon]}
all = pear, kiwi, fig, plum, lemon
t = 5050
OptionalDouble[50.5] OptionalDouble.empty
{3=1, 4=3, 5=1}
20
first p: pear
Optional[27]
```

`reduce(0, Integer::sum)` reduces the elements to a single value. The initial value is 0, and each element is added to the previous result: 0 + 4 + 8 + 15 + 16 + 23 = 66. `Collectors.groupingBy(String::length)` groups the fruits by length and produces a `Map<Integer, List<String>>`. `Collectors.joining(", ")` joins the strings with a separator in between. `IntStream.rangeClosed(1, 100)` is a stream of the `int` values from 1 to 100. Primitive streams (`IntStream`, `LongStream`, `DoubleStream`) do no boxing and offer ready-made terminal operations such as `sum` and `average`; the total is 5050.

The return type of `findFirst` is `Optional<String>`. `Optional` is a container that expresses the "there may be no result" situation explicitly instead of with `null`. No fruit starts with "z"; `orElse("none")` returns the given default for an empty container. `ifPresent` does something only if there is a value. The extra lines show three more things. `average` returns an `OptionalDouble`, because the average of an empty stream is undefined. `counting()` can be given to `groupingBy` as a second collector. `mapToInt` moves from an object stream to a primitive stream. Calling `get()` on an empty `Optional` throws `NoSuchElementException: No value present`. That is why `orElse` or `ifPresent` is preferred to `get`.

5.8.6 Side effects and parallel streams

Streams describe "what to do"; the lambdas in the pipeline are expected not to change the outside world. A lambda that modifies an outside variable or collection is called **side-effecting**. The result should be produced with `map`, `collect`, and `reduce`. This rule becomes mandatory in parallel streams. `parallel()` or `parallelStream()` splits the work among the threads of the common `ForkJoinPool`.

```
List<Integer> list = new ArrayList<>();
IntStream.range(0, 10000).parallel()
    .forEach(list::add);          // WRONG: shared list

List<Integer> safe = IntStream.range(0, 10000).parallel()
    .boxed()
    .collect(Collectors.toList()); // RIGHT: 10000 items
```

In the first pipeline, several threads call `add` on the same `ArrayList` at the same time. `ArrayList` is not thread-safe. Two threads can write to the same cell, or one can write to the old array while another is growing the internal array. The result varies from run to run. This note's measurement program ran the first pipeline 200 times on a machine with 12 cores:

```
200 trials: AIOOBE 58, short 91 (min 638, max 10000), exact 51, other 0; cores 12
200 trials: AIOOBE 51, short 145 (min 969, max 10000), exact 4, other 0; cores 12
```

All three outcomes were seen: `ArrayIndexOutOfBoundsException`, fewer than 10,000 elements, and sometimes, by chance, the right size. That the correct result appears now and then makes the bug even more dangerous; a test may not catch it. The second pipeline gives 10,000 elements and the correct order on every run. `collect` gives each thread its own intermediate list and merges them safely at the end.

```
size 6805
correct 10000 0 9999
```

A parallel stream is not always fast. Splitting the work, distributing it to threads, and merging the results has a cost. On small data, with cheap operations, with jobs that wait for I/O, and with order-dependent operations (`sorted`, `findFirst`), a parallel stream can slow things down. Measure before moving to parallelism. Common mistakes with streams are: using a stream a second time, doing the real work inside `peek`, filling an outside collection with `forEach`, and assuming a parallel stream is automatically fast.

5.9 Choosing the right collection

5.9.1 A selection table

Choosing a collection starts with two questions: which operation will be performed most often, and does order matter? Table 5.32 gives the classes that meet typical needs.

Need	Choice	Why
Ordered list, access by index	`ArrayList`	fast `get(i)`; the default list
Frequent insertion/removal at the front/back, queue, stack	`ArrayDeque`	O(1) at both ends; faster than `Stack` and `LinkedList`
Repeatedly taking the highest-priority element	`PriorityQueue`	`poll` gives the smallest (or the highest priority by the criterion)
Unique elements, fast "is it present?"	`HashSet`	`contains` O(1) on average
Unique + insertion order	`LinkedHashSet`	order is preserved
Unique + sorted, range queries	`TreeSet`	`first`, `headSet`, `subSet`
Fast access from key to value	`HashMap`	`get` O(1) on average; the default map
Sorted keys	`TreeMap`	`firstKey`, `headMap`, `subMap`

Table 5.32: Choosing a collection by need.

If the variable is declared with an interface type (`List`, `Set`, `Map`, `Queue`, `Deque`), only the `new` line changes when the need changes. First the right interface is chosen, then the implementation.

5.9.2 Unmodifiable collections

Some collections are wanted never to change after they are created: the days of the week, a fixed list of colors, a configuration table. The factory methods introduced in Java 9 produce such collections in a single line:

```
List<String> r = List.of("red", "green");
Set<Integer> primes = Set.of(2, 3, 5);
Map<String, Integer> score =
        Map.of("Ali", 90, "Banu", 85);
System.out.println(r);
System.out.println(score.get("Banu"));
r.add("blue");
```

```
[red, green]
85
Exception in thread "main" java.lang.UnsupportedOperationException
```

`List.of`, `Set.of`, and `Map.of` return **immutable** collections. Reading is free; `add`, `remove`, and `put` throw `UnsupportedOperationException`. These collections do not accept `null` elements. A call with duplicates such as `Set.of(1, 1)` also throws `IllegalArgumentException: duplicate element: 1`.

The second way, available since Java 1.2, is `Collections.unmodifiableList`. This method returns not a new collection but a **read-only view**:

```
List<String> list = new ArrayList<>();
list.add("a");
List<String> readOnly =
        Collections.unmodifiableList(list);
list.add("b");
System.out.println(readOnly);
readOnly.add("c");
```

```
[a, b]
[a, b, x] [a, b]
Exception in thread "main" java.lang.UnsupportedOperationException
```

No change can be made through the `readOnly` view, but if the underlying `list` changes, the view changes too: when "b" is added, `readOnly` becomes `[a, b]`. In this note's program an independent copy was also taken with `List.copyOf(list)` (Java 10+). Then "x" is added to the list; the view shows it, while the copy remains `[a, b]`. To hand a list inside a class to the outside as "read only", a view is used; for a truly constant list, `List.of` or `List.copyOf` is used. Unmodifiable collections are also safe choices for hash keys and constants.

5.10 Solved problems

Example 5.1

What does the following program print? When `fibonacci(10)` returns, how many elements are in the list and what are the last three? How many times does the loop run in the call `fibonacci(1)`? Is a list really needed for this job?

```
public class Fibonacci {
    public static int fibonacci(int num) {
        List<Integer> list = new ArrayList<>();
        list.add(0);
        list.add(1);
        for (int i = 2; i < num + 1; i++) {
            list.add(list.get(i - 2) + list.get(i - 1));
        }
        return list.get(num);
    }
    public static void main(String[] args) {
        int result = fibonacci(10);
        System.out.println(result);
    }
}
```

The list starts with 0 and 1. The loop runs nine times, from `i = 2` to `i = 10`, adding the sum of the last two elements in each round. When `fibonacci(10)` returns, the list has 11 elements (indices 0…10): `[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]`. The last three are 21, 34, 55, and the program prints 55. In `fibonacci(1)`, `2 < 2` is false for `i = 2` and the loop never runs; `list.get(1)` = 1 is returned. A list is not really needed: two variables holding only the last two values are enough, and the memory drops from O(n) to O(1). This note's program also shows a limit with extra calls:

```
import java.util.*;
public class Fibonacci {
    public static int fibonacci(int num) {
        List<Integer> list = new ArrayList<>();
        list.add(0);
        list.add(1);
        for (int i = 2; i < num + 1; i++) {
            list.add(list.get(i - 2) + list.get(i - 1));
        }
        return list.get(num);
    }
    public static void main(String[] args) {
        int result = fibonacci(10);
        System.out.println(result);
        System.out.println(fibonacci(1) + " " + fibonacci(0) + " " + fibonacci(46));
        System.out.println(fibonacci(47));
    }
}
```

```
55
1 0 1836311903
-1323752223
```

`fibonacci(46)` = 1,836,311,903 is the last Fibonacci number that fits in an `int`. The 47th term exceeds the `int` limit and, because of overflow, a negative number results. The `Integer`s inside a collection are also bound by the limits of the primitive `int`; for larger values `Long` or `BigInteger` is used.

Example 5.2

What does the following `dedup` method return? What would change if `HashSet` or `TreeSet` were used instead of `LinkedHashSet`?

```
public class Dedup {
    static List<String> dedup(List<String> input) {
        Set<String> seen = new LinkedHashSet<>(input);
        return new ArrayList<>(seen);
    }

    public static void main(String[] args) {
        List<String> l = Arrays.asList("b", "a", "b", "c", "a");
        System.out.println(dedup(l));
    }
}
```

```
[b, a, c]
[a, b, c] [a, b, c]
```

`new LinkedHashSet<>(input)` adds the elements of the list to the set in order; duplicates are discarded and the order of first appearance is preserved. The result is `[b, a, c]`. `TreeSet` would give alphabetical order: `[a, b, c]`. `HashSet`, on the other hand, guarantees no order. On the second line of the program, the result of `HashSet` also came out as `[a, b, c]`; because the hash codes of one-letter strings are consecutive numbers, the buckets happened to coincide with alphabetical order. This is a coincidence and cannot be relied on. The cost of the method is O(n): one O(1)-average insertion for each element.

Example 5.3

Parenthesis balance: every opening parenthesis must be closed with the right kind and in the right order. What do `isBalanced("(a[b]c)")`, `isBalanced("(a[b)c]")`, and `isBalanced("((")` return? Why is a stack used?

```
static boolean isBalanced(String expr) {
    Deque<Character> stack = new ArrayDeque<>();
    for (char c : expr.toCharArray()) {
        if (c == '(' || c == '[') {
            stack.push(c);
        } else if (c == ')' || c == ']') {
            if (stack.isEmpty()) return false;
            char open = stack.pop();
            if (c == ')' && open != '(') return false;
            if (c == ']' && open != '[') return false;
        }
    }
    return stack.isEmpty();
}
```

An opening parenthesis is pushed onto the stack. When a closing parenthesis arrives, the opening parenthesis at the top of the stack is popped and its kind is compared. If the stack is empty or the kind does not match, the expression is unbalanced. If the stack is not empty at the end, an unclosed parenthesis remains. In "(a[b]c)", when `]` arrives `[` is on top, and when `)` arrives `(` is on top, and at the end the stack is empty: `true`. In "(a[b)c]", when `)` arrives `[` is on top: `false`. In "((", two parentheses remain on the stack at the end: `false`. This note's program also tries three additional expressions ("()", the empty string, and "[(x)](y)"):

```
true false false false true true
```

The reason a stack is used is that the most recently opened parenthesis must be the first to close. This is exactly LIFO behavior. The stack was chosen as an `ArrayDeque`, not a `Stack`. The method scans the expression once and does O(1) work per character; the total cost is O(n).

Example 5.4

What is the output of the two code fragments? (`push` = `offerFirst`, `pop` = `removeFirst`.)

```
Deque<Integer> d = new ArrayDeque<>();
d.offerLast(1);
d.offerFirst(2);
d.offerLast(3);
d.push(4);
System.out.println(d);
System.out.println(d.pollLast() + d.pop());
System.out.println(d);
```

```
Map<String, Integer> m = new TreeMap<>();
m.put("b", 1);
m.put("a", 2);
m.put("b", 3);
Set<Integer> s = new TreeSet<>(m.values());
s.add(2);
System.out.println(m + " " + s + " " + m.size());
```

```
[4, 2, 1, 3]
7
[2, 1]
{a=2, b=3} [2, 3] 2
```

The deque changes as follows: 1 → `[1]`; 2 at the front → `[2, 1]`; 3 at the back → `[2, 1, 3]`; `push(4)` at the front → `[4, 2, 1, 3]`. `pollLast()` returns 3 and `pop()` returns 4. The two `Integer`s are added with `+`, because there is no string in the expression: 7. The remaining deque is `[2, 1]`. In the second fragment, `"b"` becomes 3 with the second `put`; the map is `{a=2, b=3}`. The set built from the values is `[2, 3]`. `s.add(2)` is not added because it is a duplicate, and the size of the map is 2.

Example 5.5

How is a `mostFrequent` method, which returns the most frequent word in a `String[]` array, written with a `HashMap`?

```
static String mostFrequent(String[] words) {
    Map<String, Integer> counter = new HashMap<>();
    for (String k : words) {
        counter.put(k, counter.getOrDefault(k, 0) + 1);
    }
    String best = null;
    int maxCount = 0;
    for (Map.Entry<String, Integer> g : counter.entrySet()) {
        if (g.getValue() > maxCount) {
            maxCount = g.getValue();
            best = g.getKey();
        }
    }
    return best;
}
```

```
water
null
```

The first loop counts the words with `getOrDefault`. The second loop traverses `entrySet` and keeps the word with the largest count. For "water, tea, water, coffee, water, tea", water (3) is returned. In an empty array there are no words and the method returns `null`; the caller must check for this case. If two words occur the same number of times, which one is returned depends on the traversal order of the `HashMap`, that is, it is undefined. The cost is O(n).

Example 5.6

A `Map<String, List<Integer>>` holding each student's list of grades is built; how are the averages printed for Ali: 80, 90 and Banu: 70?

```
Map<String, List<Integer>> grades = new TreeMap<>();
grades.put("Ali", new ArrayList<>());
grades.get("Ali").add(80);
grades.get("Ali").add(90);
grades.put("Banu", new ArrayList<>(Arrays.asList(70)));
for (Map.Entry<String, List<Integer>> g : grades.entrySet()) {
    int total = 0;
    for (int n : g.getValue()) {
        total += n;
    }
    System.out.println(g.getKey() + " avg: " + total / g.getValue().size());
}
```

```
Ali avg: 85
Banu avg: 70
{Ali=[80, 90]}
```

The value of the map is a list. `grades.get("Ali")` returns that list and `.add(80)` adds directly to it; there is no need to `put` into the map again. The average is computed with integer division: 170 / 2 = 85. The last line of this note's program shows the `computeIfAbsent` pattern from Java 8. If the key is absent this method creates and inserts a new list, and if present it returns the existing one; the two steps "first `put`, then `get`" shrink into a single call.

The second exercise in the slides writes the lottery draw without `shuffle`, using a `Random` and a `TreeSet`:

```
Random r = new Random();
Set<Integer> drawn = new TreeSet<>();
while (drawn.size() < 6) {
    drawn.add(r.nextInt(49) + 1);  // 1 … 49
}
System.out.println(drawn);
```

```
[1, 32, 37, 40, 43, 45]
```

The set prevents repetition by itself. If the same number comes a second time, `add` does not add it and the loop runs one more round. `TreeSet` keeps the numbers sorted. The output differs on every run. In this method the number of loop rounds is random, but since the probability of a repeat when choosing 6 of 49 numbers is small, it finishes after a few extra rounds.

5.11 Common mistakes

This section collects the most frequent mistakes in working with collections, together with the real compiler and run-time messages.

Common mistake

A generic declaration with a primitive type. `List<int>` does not compile; the type argument must be a reference type. The correct form is `List<Integer>`; `int` values are autoboxed.

```
PrimitiveType.java:4: error: unexpected type
        List<int> grades = new ArrayList<>();
             ^
  required: reference
  found:    int
1 error
```

Common mistake

Adding an element of the wrong type. `add(42)` on a `List<String>` is rejected at compile time. If the number is to be stored as a string, `String.valueOf(42)` is added.

```
WrongType.java:5: error: incompatible types: int cannot be converted to String
        names.add(42);
                  ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full
output
1 error
```

Common mistake

Confusing `remove(int)` with `remove(Object)`. In a `List<Integer>`, `remove(1)` deletes index 1. To delete a value, `Integer.valueOf(…)` is given.

```
List<Integer> list = new ArrayList<>();
Collections.addAll(list, 10, 20, 30);
list.remove(1);                  // index 1
System.out.println(list);
list.remove(Integer.valueOf(10)); // value 10
System.out.println(list);
```

Common mistake

Comparing `Integer`s with `==`. `==` checks whether two references point to the same object. Java caches the `Integer` objects between −128 and 127, and `valueOf` always returns the same object in this range. That is why a test with 100 happens to give `true` and one with 1000 gives `false`. Value comparison is done with `equals`.

```
List<Integer> a = new ArrayList<>();
Collections.addAll(a, 1000, 1000);
System.out.println(a.get(0) == a.get(1));
System.out.println(a.get(0).equals(a.get(1)));
```

```
[10, 30]
[30]
false
true
true
```

The first two lines of the output belong to the `remove` example and the next two to the comparison with 1000. The last line is the test with 100 added to the note's program: the same comparison gives `true` in the cache range.

Common mistake

Adding to an `Arrays.asList` list. `Arrays.asList` returns a fixed-size list backed by the array. `set` works, `add` and `remove` do not. For a modifiable list, write `new ArrayList<>(Arrays.asList(…))`.

```
[9, 2, 3]
Exception in thread "main" java.lang.UnsupportedOperationException
```

Common mistake

A nonexistent index. The valid indices are 0 … `size() - 1`. In a three-element list, `get(3)` throws an exception. `remove()` on an empty queue also throws `NoSuchElementException`; `poll()`, on the other hand, returns `null`.

```
List<String> l = new ArrayList<>();
Collections.addAll(l, "a", "b", "c");
System.out.println(l.get(3));
```

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 3 out of bounds for
length 3
```

Common mistake

Sorting Turkish names with the natural order of `String`. `String` is sorted by character codes: letters such as Ç, Ö, Ş, Ü, İ, ı come after "Z". The Turkish alphabet order is given by `java.text.Collator`.

```
List<String> names = new ArrayList<>();
Collections.addAll(names,
        "Zeynep", "Çağla", "Can", "Ömer");
Collections.sort(names);
System.out.println(names);
Locale tr = Locale.forLanguageTag("tr-TR");
Collections.sort(names, Collator.getInstance(tr));
System.out.println(names);
```

```
[Can, Zeynep, Çağla, Ömer]
[Can, Çağla, Ömer, Zeynep]
[Ilgaz, irmik, ufuk, üzüm, İzmir, ılık] [Ilgaz, ılık, irmik, İzmir, ufuk, üzüm]
```

The first two lines are the output in the slides. The third line, added to the note's program, shows the extent of the problem. In the natural order, the capital "I" comes before the lowercase letters, while "İ" and "ı" fall at the very end. With the Turkish `Collator`, the order becomes Ilgaz, ılık, irmik, İzmir, ufuk, üzüm. This collator treats the case difference as secondary; "ı" and "i" are separate letters.

5.12 Summary

- The **Collections Framework** consists of three elements: interfaces (what is done), implementations (how it is done), and the algorithms in the `Collections` class. Beside the hierarchy `Iterable` → `Collection` → `List` / `Queue` / `Set`, `Map` is a separate branch and is not `Iterable`.
- Variables and parameters are declared with an **interface type**; this way the implementation changes only in the `new` line.
- **Generic types** catch the wrong type at compile time. The type argument is a reference type; primitive values are converted to wrappers by **autoboxing** and **unboxing**. Unboxing `null` throws a `NullPointerException`.
- `ArrayList` is a dynamic array: `get` / `set` are O(1), appending at the end is amortized O(1), and insertion and removal in the middle are O(n). When the capacity is full, it grows by about 1.5 times; capacity is not `size()`.
- `LinkedList` is a doubly linked list: O(1) at the ends, `get(i)` O(n). In most cases `ArrayList` is the better choice.
- `Vector` and `Stack` are old and synchronized. `ArrayDeque` is used for stacks and queues; `push` / `pop` / `peek` work at the front, `offer` at the back, `poll` at the front.
- `Queue` operations have two forms: `add` / `remove` / `element` throw exceptions, `offer` / `poll` / `peek` return `false` / `null`.
- `PriorityQueue` is a binary heap: `add` / `poll` O(log n), `peek` O(1). `println(pq)` prints the internal array and is not sorted; the elements must be `Comparable` or a `Comparator` must be given.
- `HashSet` / `HashMap` are hash tables: the bucket is computed from `hashCode` and comparison within the bucket uses `equals`; O(1) on average, no order. `LinkedHash…` preserves insertion order; `Tree…` keeps order with a red-black tree and is O(log n).
- Equal objects must have equal hash codes; a class that overrides `equals` also overrides `hashCode`. These fields of an object in a hashed collection are not modified.

- `TreeSet` / `TreeMap` offer range queries; the rule is "start included, end excluded". In `HashMap`, `put` returns the old value, and `get` returns `null` for an absent key; `Hashtable` does not accept `null`.
- The **iterator** is the mechanism behind for-each. Removal while traversing is done with `it.remove()` or `removeIf`; the list's own `remove` changes `modCount` and leads to `ConcurrentModificationException`.
- `Collections.sort` is the stable TimSort (O(n log n)). The natural order is defined with `Comparable.compareTo`, other orders with `Comparator`; in comparisons `Integer.compare` is used, not subtraction. `shuffle` is Fisher–Yates; `binarySearch` works in O(log n) on a sorted list and returns `-(insertion point) - 1` if it does not find the value.
- A **stream** is a pipeline. The pipeline has a source, lazy intermediate operations (`filter`, `map`, `sorted`), and a terminal operation that runs the pipeline (`collect`, `reduce`, `count`). Elements pass through the pipeline one at a time; a stream is single-use; lambdas must be free of side effects, especially in parallel streams.
- `List.of` / `Set.of` / `Map.of` produce an unmodifiable collection, `Collections.unmodifiableList` a read-only view, and `List.copyOf` an independent unmodifiable copy.

5.13 Key terms

collection Collections Framework generics type safety wrapper class autoboxing / unboxing interface / implementation list dynamic array linked list index stack queue priority queue deque set hashing / hash table hash code map iterator natural ordering comparator anonymous class lambda expression shuffle binary search immutable thread safety

---

5.14 Test Yourself

1. After the list `List<Integer> l = new ArrayList<>(Arrays.asList(5, 6, 7));` is created, `l.add(0, 4)`, `l.remove(Integer.valueOf(6))`, `l.remove(2)`, and `l.add(l.size(), 8)` are applied in order. What are the list and `l.indexOf(8)`?

   Answer

   `[4, 5, 6, 7]` → the value 6 is removed: `[4, 5, 7]` → index 2 (7) is removed: `[4, 5]` → 8 at the end: `[4, 5, 8]`. `indexOf(8)` is 2.

2. For an empty `Deque<String> d = new ArrayDeque<>()`, `push("x")`, `offer("y")`, `push("z")`, and `addLast("w")` are called in order. Then `String a = d.pop(); String b = d.pollLast();`

runs. What are `d` and `a + b`?

Answer

`[x]` → `[x, y]` → `[z, x, y]` → `[z, x, y, w]`. `pop` takes z from the front and `pollLast` takes w from the back. `d` = `[x, y]`, `a + b` = `"zw"`.

3. Into the queue `new PriorityQueue<Integer>(Collections.reverseOrder())`, 7, 2, 9, 4 are added. What are the results of two `poll()` calls and the remaining size?

Answer

In reverse order the largest leaves first: 9, then 7. Two elements (2 and 4) remain in the queue.

4. The characters of the string `"banana"` are counted in a `TreeMap<Character, Integer>` with `merge(c, 1, Integer::sum)`. How is the map printed?

Answer

`{a=3, b=1, n=2}`. The letters are b-a-n-a-n-a: a occurs three times, n twice, and b once; `TreeMap` prints the keys in sorted order.

5. A `TreeSet` contains the elements 15, 5, 25, 10, 20. What are `headSet(15)`, `subSet(10, 21)`, and `tailSet(26)`?

Answer

`[5, 10]` (15 excluded), `[10, 15, 20]` (10 included, 21 excluded), and `[]` (there are no elements at 26 or above).

6. What does `stream().filter(x -> x.length() > 3).map(x -> x.substring(0, 2)).sorted().collect(Collectors.toList())` produce on the list `Arrays.asList("apple", "cherry", "plum", "fig", "pear")`? What is `mapToInt(String::length).sum()`?

Answer

"fig" is eliminated; the first two letters of the rest are ap, ch, pl, pe, and once sorted the result is `[ap, ch, pe, pl]`. The sum of the lengths is 5 + 6 + 4 + 3 + 4 = 22.

7. On the list `[2, 3, 5, 8]`, `if (x == 5) numbers.remove(x);` is run inside a for-each. Is an

exception thrown? What happens to the list?

Answer

No. When 5 (index 2) is removed, `size` becomes 3; `cursor` is already 3 and `hasNext()` returns `false`. The loop ends silently and 8 is never checked. The list is `[2, 3, 8]`. The code is faulty, but this data does not reveal it.

8. What does `Collections.binarySearch(Arrays.asList(2, 4, 6, 8, 10), 9)` return?

Answer

9 is absent; it would have to go after 8 and before 10, that is, at index 4. The return value is $-(4) - 1 = -5$.

9. A `Student` class overrides only `equals`. Two objects with the same fields are added to both an `ArrayList` and a `HashSet`, and `contains` is asked with the second object. Why do the results differ?

Answer

`ArrayList.contains` uses only `equals` and returns `true`. `HashSet`, on the other hand, first finds the bucket with `hashCode`. Because the hash codes inherited from `Object` are different, the two objects fall into different buckets; the set holds two elements and `contains` returns `false`. The fix is to write `hashCode` from the same fields.

5.15 Further reading

- Oracle, *The Java Tutorials*, the "Collections" and "Aggregate Operations" trails: they explain interfaces, implementations, algorithms, and stream pipelines step by step with official examples.
- J. Bloch, *Effective Java* (3rd ed., Addison-Wesley, 2018), Items 11, 14, 45–48, and 64. It gives short, well-argued rules on `hashCode`, `Comparable`, the judicious and side-effect-free use of streams, and referring to objects by their interfaces.
- M. Naftalin and P. Wadler, *Java Generics and Collections* (O'Reilly): a detailed examination of the internal structure of generics and the collection classes and the reasons behind their design.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals* (12th ed., 2022), Chapter 9: covers collections together with the workings of hash tables, trees, and views.

5.16 References

- Oracle, *The Java Tutorials*: Collections (Interfaces, Implementations, Algorithms) and Generics.

- Oracle, *Java SE 21 API Documentation*: the `java.util` package (`Collection`, `List`, `Map`, `Collections`…) and `java.util.stream`.
- J. Gosling et al., *The Java Language Specification*, Java SE 21 Edition (4.5 Parameterized Types, 5.1.7 Boxing Conversion, 14.14.2 Enhanced for).
- P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson (Chapter 16: Generic Collections).
- K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly (Chapter 11: Data Structures).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson (Chapter 9: Collections).
- M. Naftalin and P. Wadler, *Java Generics and Collections*, O'Reilly.
- J. Bloch, *Effective Java*, 3rd ed., Addison-Wesley (Item 11: always override `hashCode` when you override `equals`; Item 64: refer to objects by their interfaces).

PART III

Robust Programs

Chapters 6–7

PART III · CHAPTER 6

Exceptions

Even a correctly written program runs into surprises from the outside world. A file it needs to read may have been deleted, a user may type letters where a number is expected, a network connection may drop. Programs also contain faults of their own: division by zero, indexing past the end of an array, using a `null` reference. Java handles all of these through a single mechanism, the exception. This chapter first shows what an exception is and what happens to a program that does not catch one. It then traces the `try`, `catch` and `finally` blocks, automatic resource closing, and nested blocks step by step. The class hierarchy of exceptions, checked and unchecked exceptions, and the `throw` and `throws` keywords form the core of the chapter. The common exception classes are illustrated one by one. The chapter ends with writing custom exception classes, reading a stack trace, and good practices.

LEARNING OUTCOMES

Concept
Define an exception, distinguish runtime problems as `Exception` and `Error`, and explain why exceptions are preferred to return codes.

Catching
Handle exceptions with `try`, `catch`, `finally` and try-with-resources, and trace the control flow step by step.

Hierarchy
Explain the relationships among `Throwable`, `Error`, `Exception` and `RuntimeException`, and why the order of `catch` blocks matters.

Types
Tell checked and unchecked exceptions apart, and apply the “catch or specify” rule.

Throwing
Throw an exception with `throw`, declare it with `throws`, and trace how an exception propagates up the call stack.

Practice
Recognize the common exception classes, write a custom exception class, and locate the source of a fault by reading the stack trace.

6.1 The idea of an exception

Programming errors fall into three groups. A compile-time error is code that breaks the rules of the language: a semicolon is missing or a variable is undeclared. The compiler finds it and the program never runs. A logic error is silent: the program runs but produces a wrong result, for example when an average is computed by dividing the sum by the wrong number. The third group consists of problems that appear while the program is running. The code compiles, but at a particular moment, with a particular value, it cannot be executed. This chapter is about that third group.

6.1.1 What is an exception?

Definition 6.1

An **exception** is an unexpected event that occurs while a program is running and disrupts its normal flow. In Java an exception is an object. It carries the type of the event, a descriptive message, and the place where the problem occurred.

Everyday examples come to mind at once. A file that is to be opened cannot be found. An integer is divided by zero. Code tries to reach the fifth element of a three-element array. A method is called through a `null` reference. None of these is something the compiler can foresee. Whether the file exists or whether the denominator is zero becomes known only while the program runs.

Two parties can create an exception. The first is the Java Virtual Machine (JVM). Situations such as division by zero or an array index out of range are noticed by the JVM itself, which creates the exception object on its own. The second is the programmer. When the programmer sees an invalid argument, for instance, the `throw` keyword deliberately creates an exception. In both cases the outcome is the same: there is an exception object and the normal flow has stopped.

The exception mechanism gives the programmer an orderly way to deal with unusual situations. Roughly speaking, when Java meets an exception it says: “I give up, I do not know what to do right now. Someone else handle this!” The code that resolves the problem does not have to be where the problem occurred.

Think of a restaurant. A waiter goes to the kitchen and learns that the requested dish has run out. The waiter cannot fix this alone and reports it to the manager. The manager either solves the problem by suggesting another dish to the customer, or passes it on to a higher level. An exception travels

between methods in the same way. If the method that notices a problem cannot solve it, it reports it to the method that called it. This chain continues until the problem reaches a method that can solve it.

6.1.2 Runtime errors: Exception and Error

When a problem arises while the program is running, the JVM throws it as a `Throwable` object. The name of this class means "something that can be thrown". `Throwable` has two main branches, and they describe very different situations.

Branch	What it describes	What the program can do	Example
Exception (`Exception`)	An unusual situation that the program can anticipate	Can catch it and recover	`10 / 0`, `arr[5]` on a three-element array, a missing file
Error (`Error`)	A serious problem at the JVM or system level	Usually not caught; the program terminates	`StackOverflowError`, `OutOfMemoryError`

Table 6.1: The two main branches of runtime problems.

The criterion behind the distinction in the table is the question "can the program recover from this situation?" If a file is not found, the program can ask the user for another file name. If the user types letters instead of a number, the program can warn and ask again. These are exceptions. If memory is exhausted or the call stack has overflowed, the program has no resources left with which to repair anything. These belong to the `Error` class.

The exception mechanism, together with the stack trace an exception leaves behind, is among the most powerful tools for finding and fixing the causes of runtime errors, that is, for debugging. The compiler reports compile-time errors, and a logic error leaves no trace. With a runtime error, however, the JVM prints the type of the problem, its message, and its exact location.

6.1.3 An uncaught exception

Let us first see what happens when no precaution is taken. The following program divides an integer by zero.

```
public class Crash {
  public static void main(String[] args) {
    int num = 10, den = 0;
    System.out.println("Start");
    int result = num / den;
    System.out.println("Result: " + result);
    System.out.println("Done");
  }
}
```

```
Start
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Crash.main(Crash.java:5)
```

The program first prints `Start`. While line 5 evaluates `num / den`, the denominator is zero. Integer division by zero has no result. The JVM creates an `ArithmeticException` object and throws it. There is no code at this line to catch the exception. The caller of `main` is the JVM itself, so the exception reaches the JVM and the program terminates. The `Result` and `Done` lines are never printed.

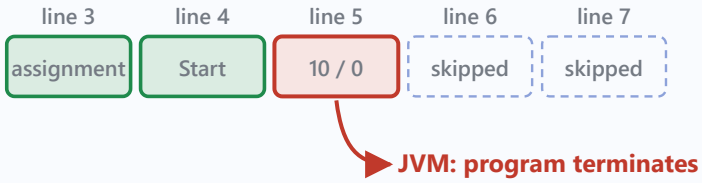


Figure 6.1: Control flow with an uncaught exception. Lines 3 and 4 run; the exception occurs at line 5 and the JVM terminates the program. Lines 6 and 7 are skipped.

The message printed by the JVM carries three pieces of information. The `Exception in thread "main"` part says that the exception was not caught in the `main` thread. Next comes the fully qualified class name of the exception: `java.lang.ArithmeticException`. The `/ by zero` after the colon is the message of the exception. The last line gives the location of the problem: the `main` method of the `Crash` class, line 5 of the file `Crash.java`. Lines beginning with `at` are called the stack trace.

Common mistake

Taking the “crash” of a program for a compile-time error. This program compiles without trouble. The compiler cannot know what value the variable `den` will have at run time. The problem appears only when the program is run. The phrase `Exception in thread` in the message shows that it is a runtime problem.

6.1.4 Handle or declare

If an exception can occur in a method, the method has two options.

The first way is to handle the exception. The method places the code that may cause the exception in a `try` block and resolves the situation itself in a `catch` block. Consider a method that reads a settings file. If the file is not found (`FileNotFoundException`), the method shows the user a message and continues with default settings.

The second way is to make the exception the caller's problem. The method does not handle the exception; it declares in its signature, with the `throws` keyword, that it may throw this exception. The exception thereby becomes the problem of the code that calls the method. The calling code faces the same choice: it either handles the exception or passes it on to the method above it.

The question that decides which way to choose is: where is the place that can resolve the exception in a meaningful way? Code that takes the file name from the user can ask again in the “file missing” case. A low-level method that reads the file line by line does not talk to the user, and it does not know where the file name came from. For that method the right choice is to declare the exception; for user-interface code the right choice is to handle it.

6.1.5 Exceptions instead of return codes

Exceptions are how a program reports that “something went wrong”. But “going wrong” is subjective. Is failing to find a name in an array an error, or an ordinary outcome? Before the exception mechanism became widespread, such situations were reported with a special return value. The following method returns `-1` if it cannot find the name it is looking for.

```
public static int indexOf(String[] names, String name) {
  for (int i = 0; i < names.length; i++) {
    if (names[i].equals(name)) {
      return i;
    }
  }
  return -1;
}
// in main
String[] names = {"Ali", "Ayşe", "Can"};
System.out.println(indexOf(names, "Can"));
int k = indexOf(names, "Ece");   // k = -1
System.out.println(names[k]);
```

```
2
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index -1 out of
bounds for length 3
    at ReturnCode.main(ReturnCode.java:14)
```

The first call finds `"Can"` at index 2 and `2` is printed. In the second call `"Ece"` is not in the array and the method returns `-1`. The calling code has forgotten to check this value. The access `names[-1]` throws an `ArrayIndexOutOfBoundsException`. The real problem arose in the search, but the error shows up somewhere else, at the array access. A fault whose source and symptom are in different places is harder to debug.

A return code has another problem: the error value can be confused with ordinary results. Here `-1` works because it is an invalid index. For a method that measures temperature, however, `-1` is a valid value, and the same method is left with no spare value to report an error.

The same method is written like this with an exception.

```
public static int find(String[] names, String name) {
  for (int i = 0; i < names.length; i++) {
    if (names[i].equals(name)) {
      return i;
    }
  }
  throw new IllegalArgumentException("Not found: " + name);
}
// in main
try {
  System.out.println(find(names, "Ece"));
} catch (IllegalArgumentException e) {
  System.out.println("Error: " + e.getMessage());
}
```

```
Error: Not found: Ece
```

If no match is found, the method throws an `IllegalArgumentException`. The message includes the name that was searched for: `Not found: Ece`. The caller can no longer forget to check. If the exception is not caught, the program stops at that point and clearly shows where the problem is. The error is not confused with a normal result: whenever the `find` method returns a number, that number is always a valid index.

For this reason special return codes such as `-1` are generally avoided and Java's exception mechanism is preferred. The benefits of exceptions can be summed up in three points:

- Error handling is explicit and consistent; the error value is not confused with a normal result.
- The code is more readable and debugging is easier: the error is visible where it occurs.
- Code is more reliable: exceptions, unlike unchecked return codes, do not slip by silently.

Common mistake

Turning every negative outcome into an exception. The result “not found” of a search method may be part of the program's ordinary flow. This is why `String.indexOf` in the Java library still returns `-1`. Exceptions are for situations that the caller does not expect and that cannot be described by an ordinary value. The decision is made according to the method's contract and is documented.

6.2 `try` and `catch`

6.2.1 The `try` and `catch` statements

The basic structure of error handling in Java is the `try` - `catch` statement. Code that may cause an error is written in the `try` block. What to do if an error occurs is written in the `catch` block.

> **Definition 6.2**
>
> The `try` **block** contains the statements that may cause an exception. The `catch` **block** runs and handles an exception of the specified type when one occurs in the `try` block; this is called *catching* the exception. The `(ArithmeticException e)` in the parentheses of `catch` is a parameter declaration: `e` is a reference to the thrown exception object.

```
try {
  // Code that may fail
  int result = 10 / 0; // ArithmeticException
} catch (ArithmeticException e) {
  // Handle the error
  System.out.println("Error: " + e.getMessage());
}
```

```
Error: / by zero
```

The expression `10 / 0` throws an `ArithmeticException`. The type of the thrown object matches the type written in `catch`. The JVM binds this object to the parameter `e` and runs the `catch` block. `e.getMessage()` returns the message of the object, the text `/ by zero`.

The flow of this structure is summarized by two rules:

- If an exception occurs, the remaining lines of the `try` block are skipped and control passes to the matching `catch` block.
- If no exception occurs, the `catch` block never runs.

In both cases the program continues its normal flow after the `catch` block. Because the exception was caught, the program does not crash.

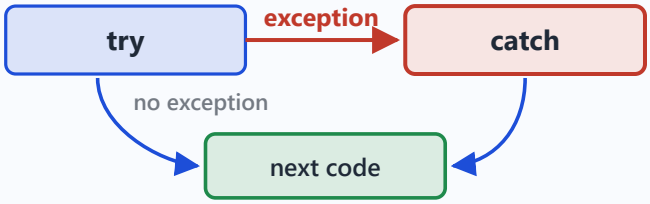


Figure 6.2: The two paths of the `try` - `catch` flow. If there is no exception, control goes straight to the next code when `try` ends. If there is one, `catch` runs first, then control goes to the next code.

The code used here is an artificial example written to demonstrate an exception. With the literal expression `10 / 0` the compiler can see the problem too and issues a warning (`division by zero`). In real programs the denominator is usually a variable whose value becomes known only at run time.

6.2.2 Code trace: the `try` - `catch` flow

This time the program from the uncaught-exception example is wrapped in a `try` - `catch` .

```
public class DivideDemo {
  public static void main(String[] args) {
    int num = 10, den = 0;
    System.out.println("Start");
    try {
      System.out.println("Before divide");
      int result = num / den;
      System.out.println("Result: " + result);
    } catch (ArithmeticException e) {
      System.out.println("Error: " + e.getMessage());
    }
    System.out.println("Done");
  }
}
```

The program advances line by line. Table 6.2 gives, for each step of the step-by-step trace in the slides, the line, the variables, and the output added at that step.

Step	Line	Call stack	Variables	New output
1	3: `int num = 10, den = 0;`	main()	num=10, den=0	—
2	4: `System.out.println("Start");`	main()	num=10, den=0	`Start`
3	6: `System.out.println("Before divide");`	main()	num=10, den=0	`Before divide`
4	7: `int result = num / den;`	main()	num=10, den=0	—
5	9: `} catch (ArithmeticException e) {`	main()	num=10, den=0, e=ArithmeticException	—

Step	Line	Call stack	Variables	New output
6	10: `System.out.println("Error: " + e.getMessage());`	main()	num=10, den=0, e=ArithmeticException	`Error: / by zero`
7	12: `System.out.println("Done");`	main()	num=10, den=0	`Done`

Table 6.2: Trace of the `DivideDemo` program.

The first two steps have nothing to do with the exception. The variables are created and the first line outside the `try` prints `Start`. In step 3 the `try` block is entered; its first line runs without trouble and `Before divide` is printed. Step 4 is the heart of the matter. While `num / den` is evaluated, the JVM creates an `ArithmeticException` object and throws it. No assignment takes place; the variable `result` never receives a value.

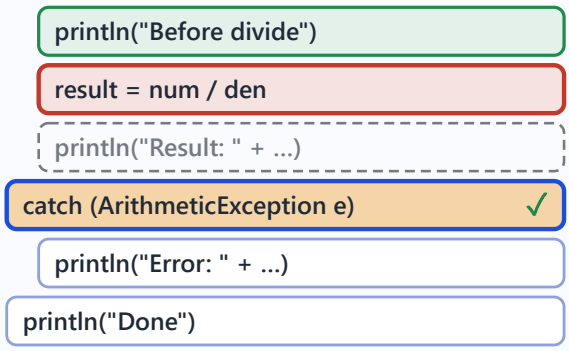


Figure 6.3: Step 5 of the trace. The first line of `try` ran successfully (green), the second threw an exception (red), the third was skipped (dashed). Because the types match, the `catch` block is running.

In step 5 the JVM looks for a suitable `catch`. The thrown object is an `ArithmeticException`, and that type is written in the `catch`. The types match and `e` refers to the exception object. The `Result` line has been skipped. In step 6, `e.getMessage()` is called and `Error: / by zero` is printed. In step 7 the `catch` block ends and the program returns to its normal flow: `Done` is printed.

Comparing the two programs is instructive. Without `try`-`catch` the program terminated at line 5 and `Done` was never printed. Now the same exception occurs, but the program runs to the end. The exception did not disappear; it was only handled.

Common mistake

Using a variable declared inside `try` outside the block. In the program above, `int result` is declared in the `try` block; its scope ends at the closing brace of the block. If `System.out.println(result)` is written after the `try`, the compiler reports `cannot find symbol`. If the variable is needed outside, it is declared before the `try` and given an initial value.

6.2.3 Getting information from an exception object

The `catch` parameter is an ordinary reference. The exception object is also an ordinary object, and it inherits its methods from the `Throwable` class. The following code shows what these methods return.

```
try {
  int[] arr = new int[3];
  arr[5] = 7;
} catch (ArrayIndexOutOfBoundsException e) {
  System.out.println(e.getMessage());
  System.out.println(e);
  System.out.println(e.getClass().getSimpleName());
  e.printStackTrace();
}
```

```
Index 5 out of bounds for length 3
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 3
ArrayIndexOutOfBoundsException
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 3
    at ExceptionDemo.main(ExceptionDemo.java:5)
```

Method	What it returns
`getMessage()`	The descriptive message only; `null` if no message was given
`toString()`	The fully qualified class name, ": " and the message; `println(e)` uses this
`getClass()`	The actual class of the exception; its name without the package via `getSimpleName()`
`printStackTrace()`	`toString()` and the stack trace; writes to `System.err`
`getCause()`	The other exception that led to this one; `null` if none

Table 6.3: Methods that give information about an exception object.

The first line of the output comes from `getMessage()`: `Index 5 out of bounds for length 3`. The second line is written by `println(e)`. `println` calls the object's `toString()` method, which joins the full class name with the message. The third line is just the short name of the class. The last two lines are produced by `printStackTrace()`. This method first writes the result of `toString()` and then the stack trace.

`printStackTrace()` writes not to standard output but to the error stream (`System.err`). Although the two streams appear on the same console, they are separate. For this reason the order of lines written to both streams may get mixed up on the console. Most integrated development environments (IDEs) show the error stream in red.

`getCause()` is for chained exceptions. A method can convert an exception it caught at a lower level into one of its own types and rethrow it. In doing so it attaches the old exception to the new object as its "cause": `new RuntimeException("Could not read settings", e)`. Code at the higher level can then reach the original source of the problem with `getCause()`. In this example the exception has no cause and the method returns `null`.

> **Common mistake**
>
> Assuming that `getMessage()` is always non-empty in a message shown to the user. For exceptions created without a message, such as `new Exception()`, `getMessage()` returns `null`. When logging, `e.toString()` or `println(e)` is safer: at least the class name is always printed.

6.2.4 Multiple `catch` blocks and their order

More than one type of exception can occur in a block of code. Consider a program that divides two numbers taken from the user. The input may not be a number, the denominator may be zero, the input may be missing. A separate `catch` block is written for each kind of error, and each block contains the reaction specific to that type.

```
try {
  // Code that may fail
} catch (ArithmeticException e) {
  // Handle ArithmeticException
} catch (NullPointerException e) {
  // Handle NullPointerException
} catch (Exception e) {
  // All other exceptions
}
```

When an exception is thrown, the `catch` blocks are tried in order from top to bottom. The first matching block runs and the others are skipped. Here "matching" means that the thrown object is of the type in the `catch` or one of its subclasses. This is the logic of the `instanceof` operator. The final `catch (Exception e)` is the general case that catches all remaining exceptions.

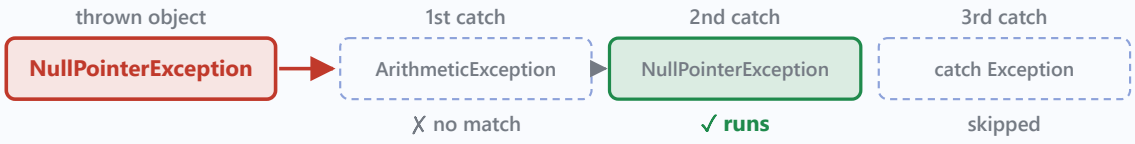


Figure 6.4: Trying the `catch` blocks for a thrown `NullPointerException`. The first block does not match, the second runs, and the third is never tried.

Order matters when there is an inheritance relationship among the `catch` types. In the following code the general type is written before the specific type.

```
try {
  int result = 10 / 0;
} catch (Exception e) {
  System.out.println("General error");
} catch (ArithmeticException e) {
  System.out.println("Arithmetic error");
}
```

```
OrderError.java:7: error: exception ArithmeticException has already been caught
    } catch (ArithmeticException e) {
```

`ArithmeticException` is an `Exception`. The first block already catches every `ArithmeticException` that is thrown, so the second block can never run. The Java compiler does not allow an unreachable `catch` block and reports an error. The rule is simple: subclasses come before superclasses. The correct order is as follows.

```
} catch (ArithmeticException e) {
  System.out.println("Arithmetic error");
} catch (Exception e) {
  System.out.println("General error");
}
```

For types with no inheritance relationship the order does not matter. `ArithmeticException` and `NullPointerException` are not subclasses of each other. An object cannot be both at once, so at most one of the two blocks matches, whichever order they are written in.

Example 6.1

`NumberFormatException` is a subclass of `IllegalArgumentException`. What happens if `catch (IllegalArgumentException e)` is written first and `catch (NumberFormatException e)` second?

A compile-time error occurs: `exception NumberFormatException has already been caught`. Since every `NumberFormatException` is an `IllegalArgumentException`, the first block already catches it. If the two blocks are swapped, the code compiles. The specific case ("not a number") is handled first, then the general case ("invalid argument").

6.2.5 Code trace: which `catch` runs?

The following method reads two numbers from an array of strings and divides one by the other. It has three separate `catch` blocks and a `finally` block. The `finally` block is the block that runs in every case, whether or not an exception occurs; here it prints one line at the end of each call.

```
static void divide(String[] input) {
  try {
    int x = Integer.parseInt(input[0]);
    int y = Integer.parseInt(input[1]);
    System.out.println("Result: " + (x / y));
  } catch (NumberFormatException e) {
    System.out.println("Not a number: " + e.getMessage());
  } catch (ArithmeticException e) {
    System.out.println("Zero: " + e.getMessage());
  } catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Missing: " + e.getMessage());
  } finally {
    System.out.println("finally ran");
  }
}
// in main
divide(new String[] {"10", "abc"});
divide(new String[] {"10", "0"});
divide(new String[] {"10"});
```

```
Not a number: For input string: "abc"
finally ran
Zero: / by zero
finally ran
Missing: Index 1 out of bounds for length 1
finally ran
```

Step	Line	Call stack	Variables	New output
1	4: `int y = Integer.parseInt(input[1]);`	main() → divide(input)	input=["10", "abc"], x=10, y=—	—
2	6: `} catch (NumberFormatException e) {` 7: `System.out.println("Not a number: " + e.getMessage());`	main() → divide(input)	input=["10", "abc"], x=10, y=—	`Not a number: For input string: "abc"`
3	12: `} finally {`	main() → divide(input)	input=["10", "abc"], x=10, y=—	`finally ran`
4	5: `System.out.println("Result: " + (x / y));`	main() → divide(input)	input=["10", "0"], x=10, y=0	—

Step	Line	Call stack	Variables	New output
5	8: `} catch (ArithmeticException e) {` 9: `System.out.println("Zero: " + e.getMessage());`	main() → divide(input)	input=["10", "0"], x=10, y=0	`Zero: / by zero`
6	4: `int y = Integer.parseInt(input[1]);`	main() → divide(input)	input=["10"], x=10, y=—	`finally ran`
7	10: `} catch (ArrayIndexOutOfBoundsException e) {` 11: `System.out.println("Missing: " + e.getMessage());` 12: `} finally {`	main() → divide(input)	input=["10"], x=10, y=—	`Missing: Index 1 out of bounds for length 1` ⏎ `finally ran`

Table 6.4: Which `catch` ran in the three calls.

In the first call the `input` array is `{"10", "abc"}`. `x` becomes 10. `Integer.parseInt("abc")` then throws a `NumberFormatException`. The division line is skipped. The JVM tries the `catch` blocks in order. The first block is for `NumberFormatException` and it matches. The other blocks are not even looked at. Then `finally` runs.

In the second call both numbers are valid: `x = 10`, `y = 0`. The division throws an `ArithmeticException`. The first `catch` does not match and the second `catch` does. The message is again `/ by zero`.

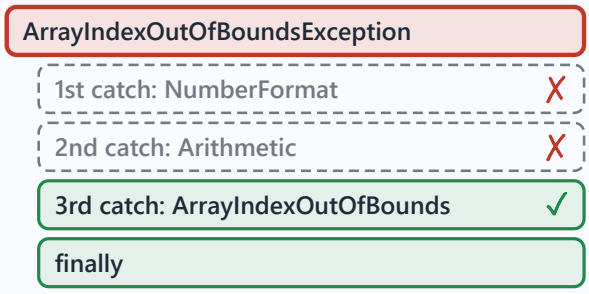


Figure 6.5: The `ArrayIndexOutOfBoundsException` thrown in the third call. The first two blocks do not match and the third does; then `finally` runs.

In the third call the array has a single element. The access `input[1]` throws an `ArrayIndexOutOfBoundsException`. The message says which index is invalid for an array of which length: `Index 1 out of bounds for length 1`. The first two `catch` blocks do not match and the third does; then `finally` runs.

In three calls three different exceptions occurred, and a separate block handled each one. What would happen if no exception occurred? In the call `divide(new String[] {"10", "4"})` no `catch` runs. The method prints `Result: 2` and `finally ran`. In integer division, 10 / 4 = 2.

6.2.6 One `catch`, several types

Sometimes different exceptions get the same reaction. Writing a separate, identical `catch` for each leads to code duplication. Since Java 7, types can be combined in a single `catch` with the `|` sign. This construct is called multi-catch.

```
try {
  Object o = "Hello";
  Integer i = (Integer) o;
} catch (ClassCastException
    | NullPointerException e) {
  System.out.println("Type or null error: "
    + e.getClass().getSimpleName());
}
```

```
Type or null error: ClassCastException
```

`"Hello"` is a `String` object; it cannot be cast to `Integer`. The thrown `ClassCastException` fits one of the two types in the block, and the block runs. `e.getClass().getSimpleName()` shows which exception was actually caught.

Multi-catch has two rules. First, in this block `e` is implicitly `final`; a new value cannot be assigned to it. If `e = null;` is written, the compiler reports `multi-catch parameter e may not be assigned`. The same assignment is not forbidden in a single-type `catch`, but it is not good practice either.

The second rule is that the alternatives cannot be subclasses of one another.

```
} catch (FileNotFoundException
    | IOException e) {
```

```
Main.java:9: error: Alternatives in a multi-catch statement cannot be related by
subclassing
    | IOException e) {
  Alternative FileNotFoundException is a subclass of alternative IOException
```

`FileNotFoundException` is a subclass of `IOException`. Writing `IOException` already covers the subtype, so writing both together is meaningless. The fix is to write only the supertype: `catch (IOException e)`.

6.3 `finally` and resource management

6.3.1 The `finally` block

A program opens a file, reads data, and closes the file. What happens if an exception occurs during reading? The rest of the `try` block is skipped; if the closing line is also inside the `try`, it never runs. The file stays open. The operating system limits the number of files a program can have open at the same time. Files, network connections and database connections that are never closed pile up, and eventually the program cannot open any new resource. This problem is called a resource leak.

> **Definition 6.3**
>
> The `finally` **block** is the block written after the `try` and `catch` blocks that runs in every case. Whether or not an exception occurs, and whether or not it is caught, `finally` runs.

```
try {
  // Code that may fail
} catch (Exception e) {
  // All other exceptions
} finally {
  // Always runs
}
```

That `finally` runs "in every case" covers four situations:

- There is no exception: `try` ends, then `finally` runs.
- There is an exception and a `catch` catches it: the `catch` ends, then `finally` runs.
- There is an exception but no `catch` matches: `finally` runs, and then the exception leaves the method.
- There is a `return` inside `try` or `catch`: `finally` runs just before the method exits.

This guarantee ensures that certain steps are certain to run. The typical use is cleanup: closing files and network and database connections, releasing locks, deleting temporary files.

A `try` must be followed by at least one `catch` or a `finally`. Writing only `try { } finally { }` is valid. In that form the exception is not caught, but cleanup is still done. A lone `try` is a compile-time error: `'try' without 'catch', 'finally' or resource declarations`.

The real exception to the rule is the call to `System.exit()`. This call stops the JVM immediately. If `System.exit(0)` runs inside `try`, `finally` does not run. A crash of the JVM itself or a shutdown of the computer also prevents `finally`, of course.

6.3.2 Code trace: `finally` and `return`

The most surprising aspect of `finally` is its relationship with `return`. In the following method both `try` and `catch` return a value.

```
static int divide(int a, int b) {
  try {
    System.out.println("try started");
    int s = a / b;
    System.out.println("try ended");
    return s;
  } catch (ArithmeticException e) {
    System.out.println("catch: " + e.getMessage());
    return -1;
  } finally {
    System.out.println("finally");
  }
}
public static void main(String[] args) {
  System.out.println("Result = " + divide(10, 2));
  System.out.println("Result = " + divide(10, 0));
}
```

```
try started
try ended
finally
Result = 5
try started
catch: / by zero
finally
Result = -1
```

Step	Line	Call stack	Variables	New output
1	15: `System.out.println("Result = " + divide(10, 2));`	main()	—	—
2	3: `System.out.println("try started");`	main() → divide(10, 2)	a=10, b=2	`try started`
3	4: `int s = a / b;` 5: `System.out.println("try ended");`	main() → divide(10, 2)	a=10, b=2, s=5	`try ended`
4	6: `return s;`	main() → divide(10, 2)	a=10, b=2, s=5	—
5	11: `System.out.println("finally");`	main() → divide(10, 2)	a=10, b=2, s=5	`finally`
6	15: `System.out.println("Result = " + divide(10, 2));`	main()	—	`Result = 5`

Step	Line	Call stack	Variables	New output
7	16: `System.out.println("Result = " + divide(10, 0));`	main()	—	—
8	3: `System.out.println("try started");`	main() → divide(10, 0)	a=10, b=0	`try started`
9	4: `int s = a / b;`	main() → divide(10, 0)	a=10, b=0	—
10	8: `System.out.println("catch: " + e.getMessage());`	main() → divide(10, 0)	a=10, b=0, e=ArithmeticException	`catch: / by zero`
11	9: `return -1;`	main() → divide(10, 0)	a=10, b=0, e=ArithmeticException	—
12	11: `System.out.println("finally");`	main() → divide(10, 0)	a=10, b=0	`finally`
13	16: `System.out.println("Result = " + divide(10, 0));`	main()	—	`Result = -1`

Table 6.5: Trace of the calls `divide(10, 2)` and `divide(10, 0)`.

In step 1 the argument of `println` must be evaluated. To do that, `divide(10, 2)` is called first. `try` starts, the division succeeds and `s = 5`. In step 4 `return s` runs. The return value 5 is prepared, but the method does not end at once. Before the method exits, the `finally` block must run. In step 5 `finally` prints. The `catch` is skipped, because there is no exception. Only in step 6 does the pending 5 return and `Result = 5` is printed.

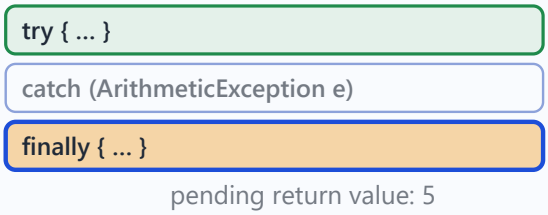


Figure 6.6: Step 5. The `try` block ended with `return s`; the return value 5 is held pending. The method returns only after `finally` has run.

The second call is `divide(10, 0)`. `try started` is printed. In step 9, `10 / 0` throws an exception. The rest of the `try`, that is, the `try ended` line and `return s`, is skipped. The `catch` matches and prints `catch: / by zero`. In step 11, `return -1` runs: the value is ready, but again `finally` runs first. At the end `-1` is returned. The conclusion is clear: in both cases `finally` ran just before the return.

Example 6.2

A method declares `int s = 1;`. Inside `try` it has `s = 2; return s;`, and inside `finally` it has `s = 3;` and `System.out.print(s + " ");`. What does `System.out.println(g())` print?

`3 2`. When `return s;` runs, the return value is computed and stored: 2. Then `finally` runs, sets `s` to 3, and prints `3 `. But the stored return value is a copy; the change to `s` does not affect it. The method returns 2. If there were also a `return` inside `finally`, that value would be returned. This usage silently erases the pending value and even a pending exception; for this reason it is discouraged.

Common mistake

Writing `return` inside `finally`. Even if `try` has thrown an exception, a `return` in `finally` destroys that exception and the method returns a value as if everything were fine. The error vanishes without a trace. `finally` is only for cleanup; it does not produce results.

6.3.3 try-with-resources

Closing a resource with `finally` is correct, but the code gets long. The resource variable is declared as `null` before the `try`, `finally` checks whether it is `null`, and then calls `close()`. Since `close()` can also throw an exception, another `try`-`catch` is often needed. Java 7 introduced a construct that shortens this work.

Definition 6.4

try-with-resources is a `try` statement that opens a resource in the parentheses after the `try` keyword. A resource opened in the parentheses is closed automatically when the block ends: the JVM calls the resource's `close()` method. Closing happens even if the block ends with an exception. The resource class must implement the `AutoCloseable` interface.

The example in the slides uses a small resource class that prints its opening and closing.

```
class Resource implements AutoCloseable {
  private String name;
  Resource(String name) {
    this.name = name;
    System.out.println(name + " opened");
  }
  void use(int x) {
    System.out.println(name + " in use");
    int s = 10 / x;
  }
  @Override
  public void close() {
    System.out.println(name + " closed");
  }
}
```

```
try (Resource r = new Resource("file")) {
  r.use(0);
} catch (ArithmeticException e) {
  System.out.println("Error: " + e.getMessage());
}
```

```
file opened
file in use
file closed
Error: / by zero
```

The order of the output deserves attention. First the resource is opened and used. The division inside `use(0)` throws an exception. At this point, before moving to the `catch`, the resource is closed: `file closed`. Only after that does the `catch` run. In other words, the resource is closed before the `catch`.

Most classes in the Java library that act as resources implement `AutoCloseable`: file streams, `Scanner`, NIO channels, database connections. With these classes there is no need to write a `finally` and a `null` check. try-with-resources is the modern and safe way to close resources instead of writing `finally`.

A class that does not implement `AutoCloseable` cannot be used in the parentheses. Even if it has a method called `shutdown()`, the compiler reports `incompatible types: try-with-resources not applicable to variable type`. The name and signature of the method the JVM will call are fixed by the interface: `close()`.

Common mistake

Forgetting the exception that `close()` declares. `AutoCloseable.close()` is declared with `throws Exception`. If a class writes `close()` keeping that declaration, try-with-resources itself can throw a checked exception. The compiler reports `unreported exception Exception` and points at the implicit `close()` call. The `Resource` class in the slides writes `close()` without `throws`, so this problem does not arise.

6.3.4 Code trace: the opening and closing order of two resources

More than one resource can be opened in the parentheses; resources are separated by semicolons.

```
try (Resource a = new Resource("file");
     Resource b = new Resource("net")) {
  b.use(0);
  System.out.println("try end");
} catch (ArithmeticException e) {
  System.out.println("Error: " + e.getMessage());
} finally {
  System.out.println("finally");
}
```

```
file opened
net opened
net in use
net closed
file closed
Error: / by zero
finally
```

Step	Line	Variables	New output
1	1: `try (Resource a = new Resource("file");`	—	`file opened`
2	2: `Resource b = new Resource("net")) {`	—	`net opened`
3	3: `b.use(0);`	—	`net in use`
4	3: `b.use(0);`	—	—
5	2: `Resource b = new Resource("net")) {`	—	`net closed`
6	1: `try (Resource a = new Resource("file");`	—	`file closed`
7	5: `} catch (ArithmeticException e) {` 6: `System.out.println("Error: " + e.getMessage());`	—	`Error: / by zero`
8	8: `System.out.println("finally");`	—	`finally`

Table 6.6: Trace of a try-with-resources with two resources.

Resources are opened in the order written: first `file`, then `net`. In step 3, `b.use(0)` first prints `net in use`, and then the `10 / 0` exception is thrown. The `try end` line is skipped. But control does not pass to the `catch` at once; first the resources are closed.

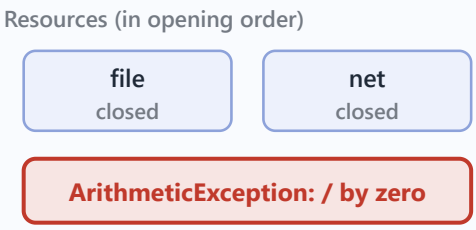


Figure 6.7: Step 6. Both resources were closed despite the exception. The exception object has not been caught yet; the `catch` comes next.

Closing is done in reverse order. The most recently opened `b` is closed first, then `a`. The reason is the possible dependency between resources. The second resource often uses the first; for example, a reader wraps a file stream. The outer wrapper is closed first, then the inner stream. After both resources are closed the `catch` runs, and finally the `finally`. The order is summarized as: open, use, close in reverse order, `catch`, `finally`.

What would the order be without an exception? In the call `b.use(5)` the division succeeds. `try end` is printed, then the resources are again closed in reverse order and `finally` runs:

```
file opened
net opened
net in use
try end
net closed
file closed
finally
```

6.3.5 Nested `try` blocks

A `try` block can contain another `try` block. This structure is used to take a precaution specific to one part of a task while taking a general precaution for the whole task.

```
try {
  // Outer try block
  try {
    // Inner try block
    // Code that may fail
  } catch (Exception e) {
    // Inner catch block
  } finally {
    // Inner finally block
  }
} catch (Exception e) {
  // Outer catch block
} finally {
  // Outer finally block
}
```

If an exception occurs in the inner `try`, the JVM proceeds in this order:

1. The `catch` blocks of the inner `try` are tried. If there is a suitable `catch`, it runs.
2. If there is no suitable `catch`, the exception moves outward. The `catch` blocks of the outer `try` are tried and the first suitable one runs.
3. If no `catch` matches, the exception leaves the method. If nobody finally catches it, the JVM's default handler terminates the program.

In every case, the `finally` block at each level the exception passes through runs.

6.3.6 Code trace: the exception propagates outward

```
public static void main(String[] args) {
  try {
    System.out.println("Outer try");
    try {
      System.out.println("Inner try");
      String s = null;
      System.out.println(s.length());
    } catch (ArithmeticException e) {
      System.out.println("Inner catch");
    } finally {
      System.out.println("Inner finally");
    }
    System.out.println("Outer try cont.");
  } catch (NullPointerException e) {
    System.out.println("Outer catch: NPE");
  } finally {
    System.out.println("Outer finally");
  }
  System.out.println("Program goes on");
}
```

```
Outer try
Inner try
Inner finally
Outer catch: NPE
Outer finally
Program goes on
```

Step	Line	Call stack	Variables	New output
1	3: `System.out.println("Outer try");`	main()	—	`Outer try`
2	5: `System.out.println("Inner try");` 6: `String s = null;`	main()	s=null	`Inner try`
3	7: `System.out.println(s.length());`	main()	s=null	—
4	8: `} catch (ArithmeticException e) {`	main()	s=null	—
5	11: `System.out.println("Inner finally");`	main()	s=null	`Inner finally`
6	14: `} catch (NullPointerException e) {` 15: `System.out.println("Outer catch: NPE");`	main()	e=NullPointerException	`Outer catch: NPE`
7	17: `System.out.println("Outer finally");`	main()	—	`Outer finally`
8	19: `System.out.println("Program goes on");`	main()	—	`Program goes on`

Table 6.7: Trace of the nested `try` program.

The outer `try` starts, then the inner `try`. The variable `s` receives the value `null`. In step 3, `s.length()` is called. A method cannot be called through a `null` reference; the JVM throws a `NullPointerException`. The inner `catch` catches only `ArithmeticException` and does not match. Before the exception leaves the inner level, the inner `finally` runs: `Inner finally`.

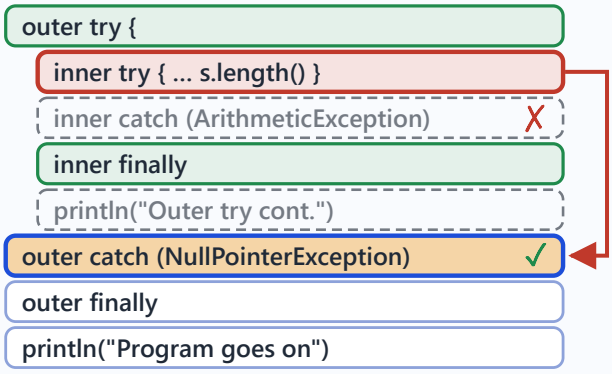


Figure 6.8: Step 6. The exception left the inner `try`; the inner `catch` did not match and the inner `finally` ran. The line `Outer try cont.` was skipped and the outer `catch` matched.

In step 6 the exception has moved to the outer level. In the outer `try`, the `Outer try cont.` line that follows the inner block is skipped. The outer `catch` is for `NullPointerException` and it matches. Then the outer `finally` runs. Because the exception has been handled, the program returns to its normal flow and `Program goes on` is printed.

Example 6.3

Let the inner `try` run `Object o = "12"; Integer i = (Integer) o;`. Let the inner `catch` catch `NumberFormatException` and the outer `catch` catch `ClassCastException`. Let there be a `finally` at each level and, at the end of the outer `try`, a line that prints `outer try end`. What does the program print?

`inner finally`, `outer catch`, `outer finally` (then the last line of the program prints `end`). `o` is a `String` object. Although `"12"` looks like a number, the cast does not convert the text to a number; it throws a `ClassCastException`. The inner `catch` does not catch this type. As the exception moves outward, the inner `finally` runs. `outer try end` is skipped, the outer `catch` catches, the outer `finally` runs, and the program continues.

6.4 The exception hierarchy

6.4.1 The `Throwable` tree

In Java the exception classes form a class hierarchy. At the root of the tree is the class `Throwable`. `Throwable` is the ancestor of everything that can be thrown. Only objects of `Throwable` and its subclasses can be thrown with `throw` and caught with `catch`. A line such as `throw "error";` therefore does not compile: `incompatible types: String cannot be converted to Throwable`.

`Throwable` has two direct subclasses: `Error` and `Exception`. `RuntimeException`, one of the subclasses of `Exception`, forms a family of its own. Figure 6.9 shows this structure.

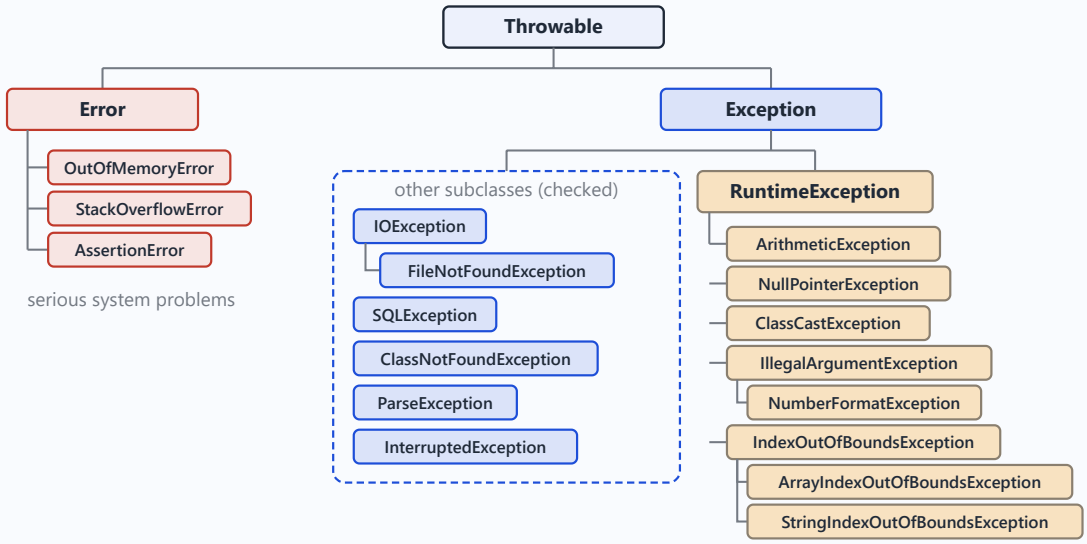


Figure 6.9: The exception class tree. Blue boxes are checked exceptions. Orange boxes are `RuntimeException` and its subclasses; red boxes are `Error` and its subclasses. These two families make up the unchecked exceptions.

The hierarchy has a practical consequence: a `catch` block catches the type written in it and all of its subclasses. This is a natural consequence of inheritance. Every `FileNotFoundException` is also an `IOException`. So `catch (IOException e)` catches both `FileNotFoundException` and other input/output errors. Exceptions of the same family can be caught with a single supertype.

A few relationships in the tree are worth noting separately. `NumberFormatException` is a subclass of `IllegalArgumentException`. `ArrayIndexOutOfBoundsException` and `StringIndexOutOfBoundsException` are subclasses of `IndexOutOfBoundsException`. `StackOverflowError` and `OutOfMemoryError` derive not directly from `Error` but from its subclass `VirtualMachineError`. The figure omits this intermediate level for simplicity.

Java also supports writing new exception classes. An application-specific situation, for example "insufficient balance", is added to this hierarchy with a class of its own. Such classes are attached to a branch of the tree and obey the same rules.

Example 6.4

`new FileReader("missing.txt")` throws a `FileNotFoundException`. Which of the types `RuntimeException`, `Error`, `SQLException` and `IOException` catches this exception?

Only `IOException`. The superclass chain of the exception is `FileNotFoundException` → `IOException` → `Exception` → `Throwable`. `RuntimeException` and `SQLException` are separate branches of `Exception`. `Error` is not in the `Exception` family at all. `catch (Exception e)` or `catch (Throwable t)` would also catch it, but they are not among the options.

6.4.2 `Error` and `Exception`

The two big branches of the tree answer different questions.

Definition 6.5

`Error` indicates that the program has met a serious problem from which it cannot recover. Coping with these situations is mostly impossible and the program usually terminates. `Exception` represents situations that the program can anticipate and can often recover from; this is the type that is expected to be handled with `try` - `catch`.

Examples of `Error` are extreme situations: the program running out of memory (`OutOfMemoryError`), the call stack overflowing (`StackOverflowError`), a disk drive "disappearing". These situations arise less from a line of the program than from the state of the environment. Catching `Error` in application code is not recommended. If memory is exhausted, the `catch` block itself may have no memory left to run.

Examples of `Exception`, on the other hand, are everyday problems: the file is missing, the network dropped, the input is invalid, the reference is `null`. Some of these come from outside the program; some are the programmer's mistake. `RuntimeException`, a subclass of `Exception`, mostly represents programmer errors. The distinction between checked and unchecked exceptions is based on this difference.

`Error` and `Exception` both derive from `Throwable`; so both can be thrown and, technically, both can be caught. But the one that "should be handled" is the `Exception` family.

Common mistake

Believing that "`catch (Exception e)` catches everything". `Exception` does not include the `Error` family. A `StackOverflowError` caused by infinite recursion is not caught by this block and the program terminates with a stack trace. The type that catches everything is `Throwable`; but catching it is not recommended either, because it hides `Error`s.

6.5 Checked and unchecked exceptions

6.5.1 Two kinds of exceptions

Java exceptions fall into two groups according to the compiler's attitude.

Definition 6.6

A **checked exception** is an exception that the compiler verifies at compile time. The programmer must either handle or declare this exception. Classes that derive from `Exception` but not from `RuntimeException` are in this group. An **unchecked exception**, on the other hand, is not verified at compile time and arises at run time. `RuntimeException`, `Error` and their subclasses are in this group.

Checked exceptions describe situations known in advance to be possible. Every program that reads a file must face the possibility that the file is not found. The compiler asks for a plan for this possibility. Unchecked exceptions, on the other hand, are mostly the symptom of a programming error. If the compiler demanded a plan at every division, every array access and every method call, the code would become unreadable.

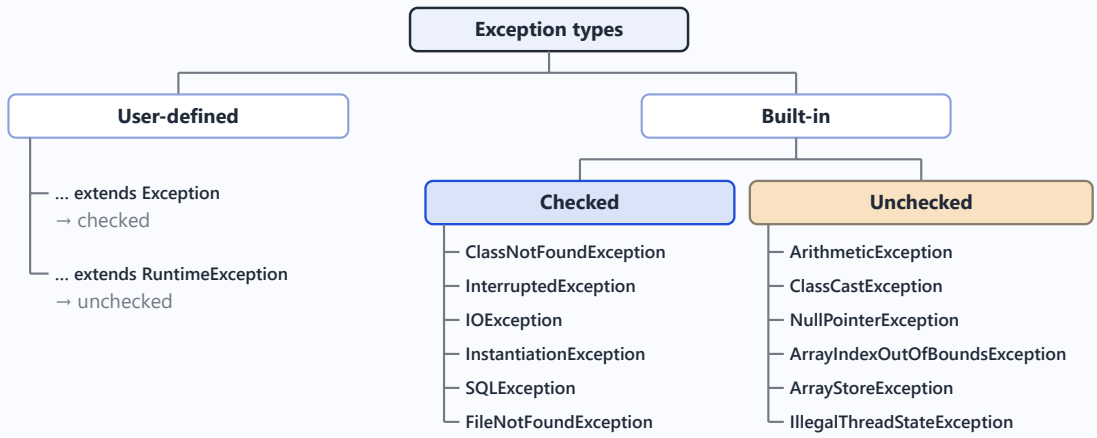


Figure 6.10: Exception types. Built-in exceptions are divided into checked and unchecked. A user-defined exception is checked if it extends `Exception` and unchecked if it extends `RuntimeException`.

6.5.2 Checked exceptions: catch or declare

Java's rule for checked exceptions is the “catch or declare” rule. Code that may throw an exception is either handled inside a `try` - `catch` or declared in the method's signature with `throws`. This rule is verified at compile time; if it is not obeyed, the program does not compile.

The rationale is simple. However correctly a program that tries to read a file is written, the file may have been deleted in the meantime. This is not a programmer's mistake; it is a fact of the outside world. The compiler makes sure the programmer does not forget this possibility.

```
public void readFile() {
  FileReader reader = new FileReader("data.txt");
}
```

```
Main.java:5: error: unreported exception FileNotFoundException; must be caught or declared
to be thrown
  FileReader reader = new FileReader("data.txt");
```

The constructor of `FileReader` declares `throws FileNotFoundException` in its signature. The method `readFile` neither catches nor declares this exception. The compiler gives the two ways to fix the error in the message itself: *must be caught* or *declared to be thrown*. The fix by declaring is as follows:

```
public void readFile() throws FileNotFoundException {
  // Read the file
  FileReader reader = new FileReader("data.txt");
}
```

Now the method compiles. But the problem has not disappeared; it has passed to the code that calls the method. Every piece of code that calls `readFile()` must obey the same rule.

6.5.3 Handling a checked exception

The second solution is to catch the exception inside the method.

```
public static void readFile2() {
  try {
    FileReader reader = new FileReader("data.txt");
    // Read the file
  } catch (FileNotFoundException e) {
    // Checked exception handled here
    System.err.println("File not found: " + e.getMessage());
    // Rethrow if needed:
    // throw e;
  }
}
```

```
File not found: data.txt (The system cannot find the file specified)
```

In this version the method catches the exception itself; it has no `throws` in its signature. The calling code needs to do nothing. The message is written to `System.err`, the error output stream. The second part of the message comes from the operating system. The output above was obtained on Windows: `(The system cannot find the file specified)`. On Linux the same situation prints `(No such file or directory)`.

The `throw e;` in the comment shows another possibility. The caught exception can be rethrown, for example after logging it. Then the exception again leaves the method, and the method signature must again declare `throws FileNotFoundException`.

Common mistake

Trying to catch a checked exception that is never thrown. If `catch (FileNotFoundException e)` is written for a `try` that contains no file operation, the compiler reports `exception FileNotFoundException is never thrown in body of corresponding try statement`. The compiler knows where checked exceptions can come from. `catch (Exception e)` does not produce this error, because `Exception` also covers the unchecked `RuntimeException`s.

6.5.4 Unchecked exceptions

Unchecked exceptions need not be handled or declared. They are usually the result of logic errors or programmer mistakes: division by zero, use of `null`, exceeding an array bound. They do not have to be listed with `throws` or wrapped in `try`-`catch`; the compiler stays silent.

```
public void divide(int num, int den) {
  // Unchecked: ArithmeticException
  int result = num / den;
}
```

This method compiles and does not need `throws`. If `den` is zero, an `ArithmeticException` occurs at run time. The right fix for such an error is often not to catch it but to prevent it. The denominator is checked before dividing, and `null` is checked before use.

```
public void divide(int num, int den) {
  if (den == 0) {   // prevention
    System.out.println("Divisor cannot be 0");
    return;
  }
  int result = num / den;
}
```

In the prevention version the exception never occurs. The check catches the problem at its source and lets the programmer decide how to handle the situation. Here the method prints a message and returns early. In a real program it is often more correct to throw an `IllegalArgumentException` with a meaningful message, because that clearly tells the caller that it made a mistake.

6.5.5 Comparing the three families

	Checked	Unchecked	`Error`
Superclass	`Exception` (except `RuntimeException`)	`RuntimeException`	`Error`
Does the compiler check?	Yes: handle or declare	No	No
Typical cause	Conditions outside the program (file, network, database)	Programmer error (`null`, bounds, division)	JVM or system problem
What to do	Catch and recover, or pass on with `throws`	Fix the code, check the input	Usually do not catch; terminate the program cleanly
Examples	`IOException`, `SQLException`, `ClassNotFoundException`	`NullPointerException`, `ArithmeticException`	`OutOfMemoryError`, `StackOverflowError`

Table 6.8: Checked exceptions, unchecked exceptions and `Error`.

Example 6.5

Let us classify `IOException`, `NumberFormatException`, `SQLException`, `NullPointerException`, `FileNotFoundException`, `ClassCastException` and `StackOverflowError` as checked or unchecked.

For each class, look at the superclass chain. If `RuntimeException` or `Error` is in the chain, the class is unchecked. Checked: `IOException`, `SQLException`, `FileNotFoundException`. Unchecked: `NumberFormatException` (via `IllegalArgumentException`), `NullPointerException`, `ClassCastException`, and `StackOverflowError`, which is an `Error`.

Example 6.6

Written inside `main` with no `try-catch` and no `throws`, which of the lines `Integer.parseInt("5")`, `Thread.sleep(100)`, `new ArrayList<Integer>(-1)` and `Objects.requireNonNull(x)` does not compile?

`Thread.sleep(100)`. This method declares that it may throw the checked `InterruptedException`; the compiler reports `unreported exception InterruptedException`. The others throw at most an unchecked exception (`NumberFormatException`, `IllegalArgumentException`, `NullPointerException`). The compiler does not check these. The third line compiles but throws an exception when it runs.

6.6 Throwing exceptions

6.6.1 The `throw` statement

When the Java runtime meets an unexpected situation, it creates an exception object and throws it. The aim is to report the situation to the code at a higher level. The programmer can do the same with the `throw` keyword.

Definition 6.7

The `throw` **statement** throws an exception object: `throw expression;`. The value of the expression must be a `Throwable` object. The object is usually created on the same line with `new`, and a message is given to its constructor.

```
throw new Exception("This is an exception!");
```

An exception changes the normal flow of the program. The lines after `throw` do not run; control passes to a suitable `catch`. The thrown exception can be caught in the calling code with a `try-catch`. If it is handled nowhere, the JVM terminates the program and reports what went wrong and where.

```
try {
  throw new Exception("This is an exception!");
} catch (Exception e) {
  System.out.println(e);
}
```

```
java.lang.Exception: This is an exception!
```

There are many types that can be thrown: `IOException`, `NullPointerException`, `IllegalArgumentException` and custom classes written by the programmer. In the example above, `println(e)` prints the result of the object's `toString()`: the full name of the class and the message given to the constructor.

Common mistake

Writing code after `throw` in the same block. If `throw` runs unconditionally, the line after it can never be reached. The compiler notices this and reports `unreachable statement`. `throw` is usually inside an `if`; in that case the following lines run when the condition is false.

6.6.2 The type that best describes the situation

When choosing an exception type, choose the class that best represents the situation. `IllegalArgumentException` suits an invalid argument, and `IllegalStateException` suits an operation that cannot be done in the object's current state. If none of the existing classes fits, a program-specific exception class is written.

```
class CustomException extends Exception {
  public CustomException(String message) {
    super(message);
  }
}
public static void check(int value)
    throws CustomException {
  if (value < 0) {
    // Check a condition and throw
    throw new CustomException("Value cannot be negative!");
  }
  // Value is OK: carry on
  System.out.println("Value OK: " + value);
}
// in main
try {
  check(5);
  check(-3);
  check(8);
} catch (CustomException e) {
  System.out.println("Caught: " + e.getMessage());
}
```

```
Value OK: 5
Caught: Value cannot be negative!
```

`CustomException` is a subclass of `Exception`. Because it does not derive from `RuntimeException`, it is a checked exception. So `throws CustomException` is mandatory in the signature of the `check` method. If this declaration is removed, the compiler reports `unreported exception CustomException; must be caught or declared to be thrown`.

The output has no `Value OK: 8` line. The moment `check(-3)` throws an exception, the rest of the `try` is skipped; `check(8)` is never called.

6.6.3 Code trace: `throw`

Let us trace the same program step by step together with the call stack. The call stack is the stack that holds the frames of the running methods; each method call pushes a frame onto the stack, and the frame is removed when the method ends.

```
static void check(int value) throws CustomException {
  if (value < 0) {
    throw new CustomException("Value cannot be negative!");
  }
  System.out.println("Value OK: " + value);
}
// in main
try {
  check(5);
  check(-3);
  check(8);
} catch (CustomException e) {
  System.out.println("Caught: " + e.getMessage());
}
```

Step	Line	Call stack	Variables	New output
1	9: `check(5);`	main()	—	—
2	2: `if (value < 0) {`	main() → check(5)	value=5	—
3	5: `System.out.println("Value OK: " + value);`	main() → check(5)	value=5	`Value OK: 5`
4	10: `check(-3);`	main()	—	—
5	2: `if (value < 0) {`	main() → check(-3)	value=-3	—
6	3: `throw new CustomException("Value cannot be negative!");`	main() → check(-3)	value=-3	—
7	12: `} catch (CustomException e) {`	main()	e=CustomException	—
8	13: `System.out.println("Caught: " + e.getMessage());`	main()	e=CustomException	`Caught: Value cannot be negative!`
9	14: `}`	main()	—	—

Table 6.9: Trace of the `throw` program.

In step 1, `main` calls `check(5)` inside the `try`, and a new frame is pushed onto the stack. `5 < 0` is false; the `throw` line is skipped. `Value OK: 5` is printed and the method returns normally. In step 4 the next call is `check(-3)`. This time the condition is true.

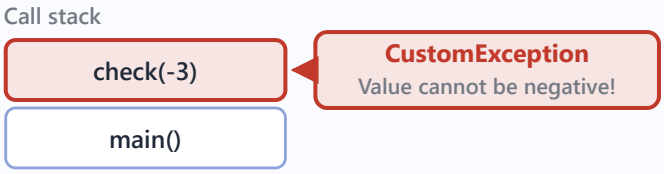


Figure 6.11: Step 6. A `CustomException` object was created and thrown in the frame of `check(-3)`. Because the method has no `catch`, this frame will be discarded.

In step 6 the exception object is created with `new` and `throw` throws it. Line 5 of the method does not run. `check` has no `catch`; so its frame is discarded and the exception passes to `main`. In step 7 the rest of the `try`, that is `check(8)`, is skipped and a suitable `catch` is found. In step 8, `getMessage()` returns the message given to the constructor: `Caught: Value cannot be negative!`. In the last step the `catch` ends and the program returns to its normal flow.

6.6.4 The difference between `throw` and `throws`

The two keywords differ by a single letter, but their jobs are entirely different.

	`throw`	`throws`
Purpose	Throws an exception (now)	Declares the exceptions a method may throw (might happen later)
Where	In the method body, as a statement	In the method signature, after the parameters
Followed by	A single object: `throw new X("…");`	One or more types: `throws X, Y`
Example	`throw new ArithmeticException("Division by zero");`	`void divide(int x, int y) throws ArithmeticException`

Table 6.10: Comparison of `throw` and `throws`.

`throw` is an action; `throws` is a warning label. A method may never throw an exception that it declares with `throws`; the declaration only announces a possibility. Declaring unchecked exceptions with `throws` is not mandatory, but it can be written for documentation. The `throws ArithmeticException` in the table is a declaration of this kind.

Confusing the two words leads to two typical compile-time errors. In the first, `throws` is used in the body:

```
if (x < 0) {
  throws new IllegalArgumentException("negative");
}
```

```
Main.java:6: error: illegal start of expression
  throws new IllegalArgumentException("negative");
```

There is no statement called `throws` in a body; the compiler cannot recognize the line as the start of an expression. In the second, a checked exception is thrown without being declared:

```
static void f(int x) {
  if (x < 0) throw new Exception("negative");
}
```

```
Main.java:5: error: unreported exception Exception; must be caught or declared to be
thrown
  if (x < 0) throw new Exception("negative");
```

`Exception` is a checked type because it is not a `RuntimeException`. There are two fixes: add `throws Exception` to the signature, or throw an unchecked type that describes the situation better, such as `IllegalArgumentException`. Because a negative argument is a programmer error, the second fix is more appropriate.

Common mistake

Believing that "`throw` can be used only in a `catch` block". `throw` is a valid statement anywhere in a body: inside an `if`, in a loop, in a `catch`, even on the first line of a method. When used inside a `catch` it rethrows the caught exception.

6.6.5 Calling methods that throw exceptions

When a method is called, the exceptions declared in its signature are taken into account and a suitable strategy is chosen. The file-reading method in the slides declares a checked exception.

```
public void readFile() throws FileNotFoundException {
  // Read the file
}
```

The first strategy is for the caller to handle the exception. The call is written inside a suitable `try` - `catch` block. This is the preferred way in most cases.

```
try {
  readFile();
} catch (FileNotFoundException e) {
  // Handle the missing file
  System.err.println("File not found: " + e.getMessage());
}
```

The second strategy is for the caller to declare the exception too. The calling method declares the same type with `throws` in its signature, and the problem passes to a higher level.

```
public void makeReport() throws FileNotFoundException {
  readFile();
}
```

The second way is less preferred than catching because it merely postpones the problem. Still, it is sometimes the right choice: if `makeReport` does not talk to the user, it cannot decide what to do when the file is not found either. The type in the declaration can be the thrown type itself or one of its superclasses. Writing `throws IOException` also compiles, because every `FileNotFoundException` is an `IOException`. If no strategy is chosen, the compiler reports `unreported exception FileNotFoundException` for the line `readFile();`.

Even the `main` method can write `throws`: `public static void main(String[] args) throws IOException`. Then the exception is passed all the way to the JVM, and the uncaught exception terminates the program. This is a practical shortcut in small experimental programs; in real applications it is more correct to give the user an understandable message.

6.6.6 Code trace: exception propagation

Definition 6.8

Exception propagation is the movement of an uncaught exception down the call stack toward the calling method. Every method the exception passes through ends at that moment, and its frame is removed from the stack.

```
static int divide(int a, int b) {
  System.out.println("divide started");
  int s = a / b;
  System.out.println("divide ended");
  return s;
}
static int compute(int x) {
  System.out.println("compute started");
  int r = divide(x, 0);
  System.out.println("compute ended");
  return r;
}
public static void main(String[] args) {
  try {
    int result = compute(10);
    System.out.println("Result: " + result);
  } catch (ArithmeticException e) {
    System.out.println("main: " + e.getMessage());
  }
  System.out.println("Program ended");
}
```

```
compute started
divide started
main: / by zero
Program ended
```

Step	Line	Call stack	Variables	New output
1	15: `int result = compute(10);`	main()	—	—
2	8: `System.out.println("compute started");`	main() → compute(10)	x=10	`compute started`
3	9: `int r = divide(x, 0);`	main() → compute(10)	x=10	—
4	2: `System.out.println("divide started");`	main() → compute(10) → divide(10, 0)	a=10, b=0	`divide started`
5	3: `int s = a / b;`	main() → compute(10) → divide(10, 0)	a=10, b=0	—
6	9: `int r = divide(x, 0);`	main() → compute(10)	x=10	—
7	15: `int result = compute(10);`	main()	—	—
8	17: `} catch (ArithmeticException e) {` 18: `System.out.println("main: " + e.getMessage());`	main()	e=ArithmeticException	`main: / by zero`
9	20: `System.out.println("Program ended");`	main()	—	`Program ended`

Table 6.11: Propagation of the exception from `divide` to `main`.

In the first four steps the stack grows: `main` calls `compute(10)`, and `compute` calls `divide(10, 0)`. Now `divide` is at the top of the stack. In step 5, `10 / 0` is evaluated and the exception is born in `divide`. `divide` has no `try`; so the rest of the method does not run and its frame is discarded. `divide ended` is not printed.

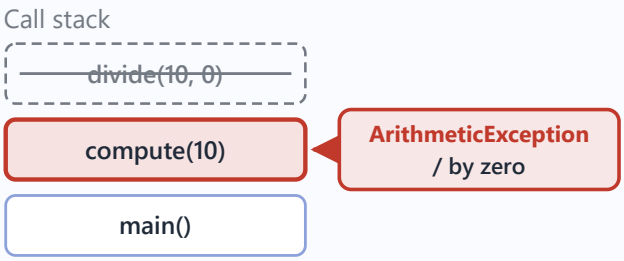


Figure 6.12: Step 6. The frame of `divide` was discarded (struck through). The exception passed to the calling method, `compute`. Because there is no `try` here either, this frame will be discarded too.

In step 6 the exception passes to the caller, `compute`. There is no `try` here either. The `compute ended` line is skipped and the frame of `compute` is discarded as well. In step 7 the exception reaches `main`. The call is inside a `try`; the JVM looks for a suitable `catch`. The `Result` line is skipped. In step 8 the `catch` matches; the exception has been caught and the propagation has stopped. In the last step the program ends normally. If nobody had caught it, the exception would have reached the JVM and the program would have terminated.

The main idea of this trace is this: the place where the exception is born and the place where it is caught can be different methods. `divide` and `compute` do not solve the problem, and they do not get involved in it. The only method concerned with the problem is `main`. Each method does only its own job; the error-handling code is gathered in one place.

Example 6.7

`c(x)` prints `c x`; if `x > 2` it throws `IllegalArgumentException("x=" + x)`, otherwise it prints `c ended`. `b(x)` calls `c(x)` and then `c(x + 2)` inside a `try` and prints `b ended`; in its `finally` it prints `b finally`. `main` calls `b(1)` inside a `try` and prints `main goes on`; the `catch` prints `main:` followed by the message; at the very end `end` is printed. What is the output?

`c 1`, `c ended`, `c 3`, `b finally`, `main: x=3`, `end`. In the first call `1 > 2` is false and `c` ends normally. In `c(3)` the exception is born; `c ended` is not printed. `b` has no `catch` but it has a `finally`: before the exception continues on its way, `b finally` is printed. `b ended` and `main goes on` are skipped.

6.6.7 How does the JVM handle an exception?

The process seen in the propagation trace works through the same steps for every exception:

1. The JVM or a `throw` statement creates an exception object. The object contains the type, the message, and the stack trace of where it was created.
2. In the method where the exception occurred, an enclosing `try` and a compatible `catch` are looked for.
3. If none is found, the method ends at that moment; its `finally` block runs, if there is one. The exception passes to the calling method and the search continues there.
4. If a suitable `catch` is found, it runs; the program continues from the line after the `catch`.
5. If `main` does not catch it either, the JVM's default handler prints the stack trace and terminates the program, or more precisely that thread.

> **Definition 6.9**
>
> The process that moves down the call stack to find a suitable `catch`, discarding the frames on the way, is called **stack unwinding**.

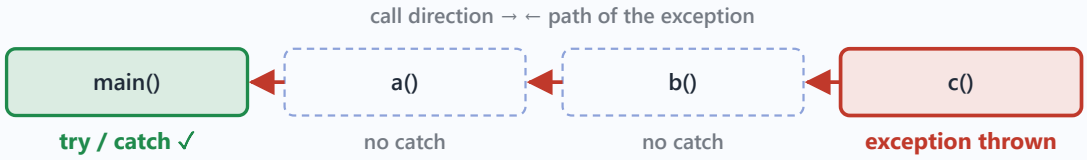


Figure 6.13: Stack unwinding. The calls were made from left to right: `main` → `a` → `b` → `c`. The exception is born in `c` and moves in the opposite direction until it finds a `catch`.

A caught exception object is an ordinary object. After the `catch` block ends, if no other reference to it is held, the object is no longer used. The garbage collector cleans it from memory after a while.

6.7 Runtime exceptions

6.7.1 The `RuntimeException` family

`RuntimeException` and its subclasses represent exceptions caused by programmer errors or logic errors. They are not checked at compile time and handling them is not mandatory. But this does not mean ignoring them. The programmer deliberately prevents or handles these situations.

Class	When it occurs	Example
`NullPointerException`	When a member of a `null` reference is used	`String s = null; s.length();`

Class	When it occurs	Example
`ArrayIndexOutOfBoundsException`	Index outside the array bounds	`int[] d = new int[3]; d[5] = 1;`
`ArithmeticException`	Integer division by zero, overflow in exact arithmetic	`10 / 0`
`IllegalArgumentException`	When an invalid argument is passed to a method	`new ArrayList<>(-1)`
`NumberFormatException`	Converting a string that is not a valid number	`Integer.parseInt("abc")`
`ClassCastException`	When an object is not of the type it is being cast to	`(Integer) obj`

Table 6.12: Commonly encountered `RuntimeException` classes.

The "line by line" examples in the slides give what happens when each line is tried *on its own*. In a real program the first exception interrupts the flow and the following lines do not run. All messages in this chapter were obtained with JDK 21, running each line inside a separate `try`.

6.7.2 `NullPointerException`

A `null` reference points to no object. Calling a method through this reference, accessing a field, or reading the `length` of an array leads to a `NullPointerException` (NPE for short). This exception is among the most common sources of bugs.

```
String str = null;
int length = str.length();
```

```
Exception in thread "main" java.lang.NullPointerException:
  Cannot invoke "String.length()" because "str" is null
```

Before Java 14 this message was just `java.lang.NullPointerException` and did not say which reference was `null`. If several references were used on the same line (`a.b().c()`) it was hard to find the culprit. Since Java 14 the helpful message says which operation could not be performed and why. The variable name appears only if the class was compiled with debugging information (`javac -g` ; IDEs do this by default). The same program compiled with plain `javac` cannot know the name and prints this:

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()"
because "<local1>" is null
```

There are several ways to prevent an NPE:

- An explicit check with `if (str != null)` .

- `Objects.requireNonNull(x, message)`: if `x` is `null`, it immediately throws an NPE with a meaningful message.
- `Objects.requireNonNullElse(x, fallback)` (Java 9+): returns the fallback value if `x` is `null`.
- The `Optional` class (Java 8+): represents the "there may be no value" case explicitly with a type.
- Libraries: for example, the `StringUtils` class of Apache Commons Lang offers `null`-safe string operations.

The following code shows these ways together.

```
1  String str = null;
2  int length = str.length();
3  Objects.requireNonNull(str, "str must not be null");
4  Optional<String> optionalStr = Optional.ofNullable(str);
5  length = optionalStr.map(String::length).orElse(0);
6  String defaultValue = Objects.requireNonNullElse(str, "0");
7  if (str != null) {
8    System.out.println(str.length());
9  }
10 System.out.println("Ali".equals(str));
```

Line	Result
2	`NullPointerException: Cannot invoke "String.length()" because "str" is null`
3	`NullPointerException: str must not be null`
5	`length = 0`: there is no value, so `orElse(0)` takes effect
6	`defaultValue = "0"`
7–9	The condition is false; nothing is printed, no exception
10	`false`: the literal is in front, so a `null` argument is no problem

Table 6.13: Results of the lines (each line tried on its own; `str` is `null` in every case).

Line 3 throws the same exception as line 2; so what does `requireNonNull` gain? Two things. First, the programmer chooses the message. Second, and more important, the error is caught at its source. `requireNonNull` is used at the start of methods. A constructor whose parameter is `null` never creates the object. In this way a broken object does not wander around the program and throw an NPE much later in an unrelated place.

Line 5 uses `Optional`. `Optional.ofNullable(str)` produces an empty box if `str` is `null`, and otherwise a box containing `str`. `map(String::length)` takes the length of the content if the box is full and does nothing if it is empty. `orElse(0)` gives 0 if the box is empty. The result is not an NPE but 0.

Line 10 shows a common habit. `"Ali".equals(str)` is safe, because the method is called on the literal, and a literal is never `null`. `str.equals("Ali")`, on the other hand, throws an NPE when `str` is `null`.

Common mistake

Catching an NPE with `catch (NullPointerException e)` and moving on. An NPE is almost always a programming error: a variable was never assigned a value, or a method unexpectedly returned `null`. The right fix is to find where the `null` came from and correct it there. Catching only hides the symptom.

6.7.3 Out-of-bounds exceptions

The valid indexes of an array run from `0` to `length - 1`. A negative index is also invalid. Accessing an index outside these bounds throws an `ArrayIndexOutOfBoundsException`.

This class is a member of the `IndexOutOfBoundsException` family. Which class is thrown on an out-of-bounds access depends on the structure being accessed:

- an array (`int[]`), a `Vector` and the list returned by `Arrays.asList(...)` → `ArrayIndexOutOfBoundsException`;
- `ArrayList.get` → `IndexOutOfBoundsException` (the superclass itself);
- `String.charAt` → `StringIndexOutOfBoundsException`.

To catch them all at once, the common superclass is written: `catch (IndexOutOfBoundsException e)`.

```
1  int[] numbers = {1, 2, 3};
2  int value = numbers[5];
3  ArrayList<String> list = new ArrayList<>();
4  String svalue = list.get(5);
5  Vector<Integer> vector = new Vector<>();
6  value = vector.get(10);
7  String[] names = {"Alice", "Bob", "Charlie"};
8  svalue = Arrays.asList(names).get(5);
9  String str = "Hello";
10 char ch = str.charAt(10);
```

Line	Exception	Message
2	`ArrayIndexOutOfBoundsException`	`Index 5 out of bounds for length 3`
4	`IndexOutOfBoundsException`	`Index 5 out of bounds for length 0`
6	`ArrayIndexOutOfBoundsException`	`Array index out of range: 10`
8	`ArrayIndexOutOfBoundsException`	`Index 5 out of bounds for length 3`
10	`StringIndexOutOfBoundsException`	`Index 10 out of bounds for length 5`

Table 6.14: Exceptions thrown on out-of-bounds access (each line tried on its own).

Line 4 is particularly instructive. Even if the list is created with a capacity such as `new ArrayList<>(20)`, its size is 0, because no element has been added yet. The `get` method checks the index against the size, not the capacity. The message of the `Vector` in line 6 has a different form, because this old class builds its message itself. The list in line 8 is a view built on top of the `names` array; the access goes directly to the array and the array's exception is thrown.

The most common cause of an out-of-bounds access is a single character in the loop condition. The correct loop is bounded by `length`:

```
int[] numbers = {1, 2, 3};
for (int i = 0; i < numbers.length; i++) {
  System.out.print(numbers[i] + " ");
}
```

```
1 2 3
```

If the condition is written as `i <= numbers.length`, the loop runs one iteration too many. In the last iteration `numbers[3]` is accessed and an exception with the message `Index 3 out of bounds for length 3` occurs. The safest way to walk over every element of an array is the for-each loop; there no index is written at all.

6.7.4 `ArithmeticException`

`ArithmeticException` represents an error that occurs during a mathematical operation. The best-known example is dividing an integer by zero; both the `/` and `%` operators throw this exception. It has other sources too:

- The "exact" methods of the `Math` class throw on overflow: `addExact`, `multiplyExact`, `toIntExact` …
- `BigDecimal` provides high-precision arithmetic. It throws on division by zero and on a division whose result cannot be represented exactly (1/3).

Some operations, contrary to expectation, do not throw an exception:

- `double` operations do not throw: `10.0 / 0` → `Infinity`, `Math.sqrt(-1)` → `NaN`.
- `Random.nextInt(0)` throws an `IllegalArgumentException`, not an `ArithmeticException`: the bound must be positive.
- When an ordinary `int` addition overflows, no exception occurs; the value wraps around. `Integer.MAX_VALUE + 1` → `-2147483648`. This is called silent overflow. If the overflow needs to be noticed, `Math.addExact` is used.

```
1  int result = 10 / 0;
2  double dresult = Math.sqrt(-1);
3  System.out.println(dresult);
4  System.out.println(10.0 / 0);
5  int t = Math.addExact(Integer.MAX_VALUE, 1);
6  Random random = new Random();
7  int randomNumber = random.nextInt(0);
8  BigDecimal num1 = new BigDecimal("10");
9  BigDecimal num2 = BigDecimal.ZERO;
10 BigDecimal bresult = num1.divide(num2);
```

Line	Result
1	`ArithmeticException: / by zero`
3	Prints `NaN`, no exception
4	Prints `Infinity`, no exception
5	`ArithmeticException: integer overflow`
7	`IllegalArgumentException: bound must be positive`
10	`ArithmeticException: Division by zero`

Table 6.15: Arithmetic errors (each line tried on its own).

Division by zero in floating-point numbers produces special values according to the rules of the IEEE 754 standard. A positive number divided by zero gives `Infinity`, a negative number gives `-Infinity`, and `0.0 / 0` gives `NaN` (Not a Number). These values silently creep into calculations and spoil every later operation. For this reason `double` results are checked separately when necessary: `Double.isNaN(x)`, `Double.isInfinite(x)`.

The 1/3 example for `BigDecimal` was also run and tried. `BigDecimal.ONE.divide(new BigDecimal("3"))` gives this exception: `Non-terminating decimal expansion; no exact representable decimal result.` An infinite decimal expansion cannot be stored exactly. The fix is to give the division a scale and a rounding mode.

Example 6.8

Which of the expressions `10 / 0`, `10 % 0`, `Math.toIntExact(3000000000L)` and `10.0 / 0` does not throw an exception when it runs?

`10.0 / 0`. Floating-point division follows the IEEE 754 rules and produces `Infinity`. Integer division and the remainder operation with zero give `ArithmeticException: / by zero`. Because the value 3,000,000,000 does not fit in the `int` range (at most 2,147,483,647), `toIntExact` throws `integer overflow`.

6.7.5 `IllegalArgumentException`

`IllegalArgumentException` indicates that a method received an invalid argument. It is an unchecked exception. The first job of a method is to check its arguments and, if necessary, throw this exception. In this way the error is stopped at its source.

```
public void setAge(int age) {
  if (age < 0) {
    throw new IllegalArgumentException("Age cannot be negative");
  }
  this.age = age;
}
public void setName(String name) {
  if (name.length() < 3) {
    throw new IllegalArgumentException("Min 3 characters");
  }
  this.name = name;
}
```

The Java library also uses this exception widely:

- `Arrays.copyOfRange(d, 5, 2)` : the start is greater than the end;
- `new ArrayList<>(-1)` : negative capacity;
- the `setPriority(11)` method of `Thread` : the priority must be between 1 and 10;
- `new File(uri)` : the URI is not of the `file:` form.

```
1  setAge(-5);
2  setName("Al");
3  int[] numbers = {1, 2, 3};
4  Arrays.copyOfRange(numbers, 5, 2);
5  List<Integer> list = new ArrayList<>(-1);
6  File file = new File(URI.create("http://example.com/a.txt"));
7  Thread thread = new Thread(task);
8  thread.setPriority(11);
```

Line	Exception thrown
1	`IllegalArgumentException: Age cannot be negative`
2	`IllegalArgumentException: Min 3 characters`
4	`IllegalArgumentException: 5 > 2`
5	`IllegalArgumentException: Illegal Capacity: -1`
6	`IllegalArgumentException: URI scheme is not "file"`
8	`IllegalArgumentException` (no message)

Table 6.16: Examples of invalid arguments (each line tried on its own).

Not every invalid situation gives this exception. `new FileReader("invalid/file/path")` throws the checked `FileNotFoundException`. `new Thread(null)` does not throw at all: a `null` task means "nothing to do", and the thread ends immediately when started. The best-known subclass of `IllegalArgumentException` is `NumberFormatException`.

A good message states the invalid value and the expected range: `"Age cannot be negative: " + age`. The message-less exception in line 8 makes debugging harder. A programmer reading the message cannot tell what priority was given and what it should have been.

Common mistake

Forgetting `null` in argument checking. In the call `setName(null)`, `name.length()`, the check itself, throws an NPE; the meaningful message the programmer wrote is never seen. The order of checks is `null` first, then the value: `Objects.requireNonNull(name, "name must not be null")`, followed by the length check.

6.7.6 `NumberFormatException` and safe conversion

When a string is converted to a number and the string does not represent a valid number, a `NumberFormatException` occurs. `Integer.parseInt`, `Double.parseDouble`, `Float.parseFloat`, `Short.parseShort`, `Long.parseLong`, `Byte.parseByte` and their `valueOf` versions can throw this exception. The same exception occurs if the value is outside the range of the type: `Byte.parseByte("300")` gives the message `Value out of range. Value:"300" Radix:10`.

```
String strNumber = "abc";
int number = Integer.parseInt(strNumber);
strNumber = "123abc";
number = Integer.parseInt(strNumber);
String strDouble = "12.34abc";
double doubleValue = Double.parseDouble(strDouble);
String strShort = "45xyz";
short shortValue = Short.parseShort(strShort);
```

Call	Message
`Integer.parseInt("abc")`	`For input string: "abc"`
`Integer.parseInt("123abc")`	`For input string: "123abc"`
`Double.parseDouble("12.34abc")`	`For input string: "12.34abc"`
`Short.parseShort("45xyz")`	`For input string: "45xyz"`

Table 6.17: Conversion calls and `NumberFormatException` messages.

The `"123abc"` example shows the “all or nothing” rule of conversion. Although the 123 at the start of the string is valid, the string as a whole is not a number. `parseInt` does not return a partial result. `NumberFormatException` is a subclass of `IllegalArgumentException`. User input always carries this risk: either the input is checked beforehand or the exception is caught.

The conversion can be guarded in one place, in a helper method. For invalid input the method returns a default value.

```
static int safeParse(String text, int fallback) {
  try {
    return Integer.parseInt(text.trim());
  } catch (NumberFormatException e) {
    return fallback;
  }
}
// in main
System.out.println(safeParse(" 42 ", 0));
System.out.println(safeParse("forty", 0));
System.out.println(safeParse("10.5", -1));
```

```
42
0
-1
```

`trim()` removes leading and trailing whitespace. This is how the input `" 42 "` becomes 42. `parseInt(" 42 ")` alone would throw an exception, because a space is not a digit. `"forty"` is not a number; the default 0 is returned. `"10.5"` is not an `int` either; `Double.parseDouble` is used for decimal numbers.

The step-by-step trace in the slides runs the same method with three new inputs.

```
static int safeParse(String text, int fallback) {
  try {
    return Integer.parseInt(text.trim());
  } catch (NumberFormatException e) {
    return fallback;
  }
}
System.out.println(safeParse(" 7 ", 0));
System.out.println(safeParse("seven", 0));
System.out.println(safeParse("7.5", -1));
```

```
7
0
-1
```

Step	Line	Variables	New output
1	8: `System.out.println(safeParse(" 7 ", 0));`	—	—
2	3: `return Integer.parseInt(text.trim());`	text=" 7 ", fallback=0	`7`
3	9: `System.out.println(safeParse("seven", 0));`	—	—
4	3: `return Integer.parseInt(text.trim());`	text="seven", fallback=0	—
5	4: `} catch (NumberFormatException e) {` 5: `return fallback;`	text="seven", fallback=0, e=NumberFormatException	`0`
6	10: `System.out.println(safeParse("7.5", -1));`	—	—
7	3: `return Integer.parseInt(text.trim());`	text="7.5", fallback=-1	—
8	4: `} catch (NumberFormatException e) {` 5: `return fallback;`	text="7.5", fallback=-1, e=NumberFormatException	`-1`

Table 6.18: Trace of the `safeParse` method with three inputs.

In the first call the text is `" 7 "`, with one space at the start and one at the end. `trim()` removes the spaces, `"7"` is converted to a number, and 7 is returned from inside the `try`. In the second call the text is `"seven"`. `parseInt` throws an exception on a string that is not digits, and the `return` statement cannot complete.

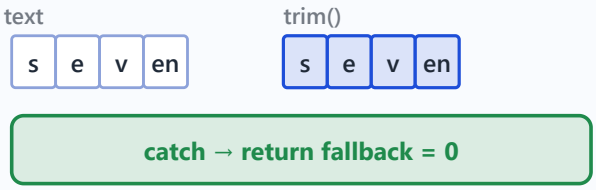


Figure 6.14: The second call. The result of `trim()` is still `seven`; the conversion fails and the `catch` block returns the default value, 0.

The `catch` matches and the default value 0 is returned. The exception does not leave the method. In the third call the text is `"7.5"` and the default value is −1. The dot is not a digit; `"7.5"` is not a valid `int`. −1 is returned. The calling code goes on its way with the default value instead of an exception.

This approach has a cost. The default value must be chosen so that it cannot be confused with a valid result. `safeParse("0", 0)` returns 0, just like `safeParse("seven", 0)`; `safeParse("-1", -1)` returns −1, just like `safeParse("7.5", -1)`. If the default value cannot be told apart from valid

inputs, it is more correct to pass the exception on to the caller. This is a small repeat of the return-code discussion.

6.7.7 `ClassCastException` and `instanceof`

If an explicit cast is made when an object is not of the type it is being cast to, a `ClassCastException` occurs. What matters is not the type of the variable but the actual type of the object. After the line `Object obj = "Hello";` the type of `obj` is `Object`, but the object it refers to is a `String`.

Raw-typed collections, that is, `ArrayList`, `Map` and `Set` written without a type parameter, accept objects of any kind. When an element of the wrong type is taken and cast, the cast fails. Sorted collections (`TreeSet`, `TreeMap`), on the other hand, also throw this exception when an element of a type that cannot be compared with the others is added.

```
 1  Object obj = "Hello";
 2  Integer number = (Integer) obj;
 3  ArrayList list = new ArrayList();
 4  list.add("Hello");
 5  number = (Integer) list.get(0);
 6  Map map = new HashMap();
 7  map.put("key", "Value");
 8  Integer value = (Integer) map.get("key");
 9  Set set = new HashSet();
10  set.add("Hello");
11  value = (Integer) set.iterator().next();
12  Object[] objArray = new String[5];
13  Integer[] intArray = (Integer[]) objArray;
```

Each of lines 2, 5, 8, 11 and 13 throws a `ClassCastException`. For example, the message of line 2 is:

```
class java.lang.String cannot be cast to class
java.lang.Integer
```

The full message in JDK 21 adds one more parenthesis: it says that both classes are in the `java.base` module and in the same loader. Line 13 is a cast between array types. `objArray` is actually a `String[]` and cannot be cast to `Integer[]`; in the message the arrays appear in the form `[Ljava.lang.String;`.

Before casting, the type of the object can be tested with `instanceof`:

```
Object obj = "Hello";
if (obj instanceof Integer) {
  Integer num = (Integer) obj;
  System.out.println("Number: " + num);
} else {
  System.out.println("obj is not an Integer");
}
```

```
obj is not an Integer
```

Since Java 16, pattern matching combines the test and the cast. If the condition is true, the variable `num` is already of type `Integer`; no separate cast line is needed.

```
if (obj instanceof Integer num) {
  System.out.println("Number: " + num);
}
```

A better solution is to move the problem to compile time. Generics tell the compiler what type of element a collection holds.

```
ArrayList<String> list = new ArrayList<>();
list.add("Hello");
Integer n = (Integer) list.get(0);
```

```
Main.java:8: error: incompatible types: String cannot be converted to Integer
Integer n = (Integer) list.get(0);
```

Instead of a runtime exception, a compile-time error is obtained. The error is found earlier, without the program ever being run. This is why generic types are preferred to raw types.

6.8 Checked exception classes

6.8.1 Commonly encountered classes

Checked exceptions represent situations that require the code to handle the error and recover appropriately. These situations mostly arise from conditions outside the program: the file system, the network, a database, another thread. The compiler makes handling these exceptions mandatory.

Class	Package	When
`FileNotFoundException`	`java.io`	The file to be opened does not exist or cannot be opened
`IOException`	`java.io`	General error in input/output operations (superclass)
`SQLException`	`java.sql`	Error in database access
`ParseException`	`java.text`	Error while parsing text according to a format
`ClassNotFoundException`	`java.lang`	A class to be loaded by name was not found
`InterruptedException`	`java.lang`	A waiting thread was interrupted (`Thread.sleep`)
`NotSerializableException`	`java.io`	An attempt was made to write an object that cannot be serialized

Table 6.19: Commonly encountered checked exceptions.

All the classes in the table derive from `Exception` and have no `RuntimeException` in their chains. `ClassNotFoundException` has one more level in between (`ReflectiveOperationException`); `NotSerializableException` is linked to `IOException` through `ObjectStreamException` .

6.8.2 `FileNotFoundException` and `IOException`

`FileNotFoundException` occurs when a file is to be opened and cannot be found. The same exception is thrown if the given path is a directory or if there is no permission to read the file.

```
try {
  FileReader fileReader = new FileReader("file.txt");
} catch (FileNotFoundException e) {
  // Handle the missing file
  System.out.println(e);
}
```

```
java.io.FileNotFoundException: file.txt
  (The system cannot find the file specified)
```

This class is a subclass of `IOException` ; `catch (IOException e)` catches it too. A relative path (`"file.txt"`) is looked up relative to the program's working directory. In development environments the working directory is usually the project folder, not the folder containing the source code. The file may be "in the wrong folder". The typical reaction to this exception is to tell the user and ask for a new name, or to continue with default values.

The message the same exception gives for a directory was also tried: `new FileReader(".")` on Windows gives `java.io.FileNotFoundException: . (Access is denied)` . Although the name of the class says "file not found", the exception is used to mean "file could not be opened".

`IOException` represents a general error in input/output operations and has many subclasses, such as `FileNotFoundException` . The following code reads a text file line by line.

```
try {
  BufferedReader reader =
    new BufferedReader(new FileReader(path));
  String line;
  while ((line = reader.readLine()) != null) {
    System.out.println(line);
  }
  reader.close();
} catch (IOException e) {
  // Handle the I/O error
  e.printStackTrace();
}
```

```
first line
second line
```

The code works, but it has a flaw. If `readLine()` throws an exception, control passes directly to the `catch` and the line `reader.close()` never runs. The file stays open: this is a resource leak. The fix is try-with-resources:

```
try (BufferedReader reader =
    new BufferedReader(new FileReader(path))) {
  String line;
  while ((line = reader.readLine()) != null) {
    System.out.println(line);
  }
} catch (IOException e) {
  System.out.println(e.getMessage());
}
```

In this version the reader is closed however the block ends. If the file does not exist, `new FileReader(path)` throws an exception; in that case no resource has been created to close, and the `catch` runs directly. `catch (IOException e)` catches both the opening error (`FileNotFoundException`) and read errors. Tried with the nonexistent `missing.txt`, the block prints the message `missing.txt (The system cannot find the file specified)`.

6.8.3 `SQLException`

`SQLException` is thrown when an error occurs during a database operation. The connection may fail, the SQL syntax may be wrong, the table may not exist, or the database driver may not be loaded. The exception object offers three methods that give details: `getMessage()`, `getSQLState()` and `getErrorCode()`.

```
try {
  Connection connection =
    DriverManager.getConnection("jdbc:mysql://localhost/school");
  PreparedStatement preparedStatement =
    connection.prepareStatement("SELECT user_id FROM users");
  // Run the query and process the results
  ResultSet resultSet = preparedStatement.executeQuery();
  while (resultSet.next()) {
    // Process the results
    System.out.println("User ID: " + resultSet.getInt("user_id"));
  }
  // Close the connection
  connection.close();
} catch (SQLException e) {
  // Handle the SQLException
  System.out.println(e);
}
```

```
java.sql.SQLException: No suitable driver
found for jdbc:mysql://localhost/school
```

This program was run without the MySQL driver on the classpath. `DriverManager` cannot find a driver that recognizes a URL starting with `jdbc:mysql:`. In this case `getSQLState()` returns `08001` and `getErrorCode()` returns 0. State codes starting with `08` are the standard codes that describe connection errors.

The `connection.close()` call in the code has the flaw seen in the `IOException` example: if an exception occurs during the query, the connection stays open. A connection is also a resource and is closed with try-with-resources. `Connection`, `PreparedStatement` and `ResultSet` are all `AutoCloseable` and can be opened in the same parentheses.

6.8.4 `ParseException` and `ClassNotFoundException`

`ParseException` (in the `java.text` package) is thrown when a string does not match the format being used to parse it.

```
try {
  SimpleDateFormat form =
    new SimpleDateFormat("dd/MM/yy");
  Date date = form.parse("01-01-2022");
} catch (ParseException e) {
  // Handle the parse error
  System.out.println(e);
}
```

```
java.text.ParseException: Unparseable date: "01-01-2022"
```

The format is `dd/MM/yy`; a `/` is expected between day and month. The text has a `-` instead. The parser reads the first two characters as the day and gets stuck at the third character. `e.getErrorOffset()` gives this position: 2, that is, the third character.

`ClassNotFoundException` is thrown when a class is to be loaded by its name (`Class.forName`) and cannot be found.

```
try {
  Class.forName("java.util.Scanner");
  System.out.println("Scanner loaded");
  // Try to load a missing class
  Class.forName("com.example.NonExistentClass");
} catch (ClassNotFoundException e) {
  System.out.println(e);
}
```

```
Scanner loaded
java.lang.ClassNotFoundException:
  com.example.NonExistentClass
```

`java.util.Scanner` is in the standard library and loads without trouble. `com.example.NonExistentClass`, on the other hand, is not on the classpath. Loading a class by name is used when working with classes not known at compile time, such as database drivers. Because the name is a string, the compiler cannot check it; whether the class will be found becomes known only at run time.

This exception should not be confused with `NoClassDefFoundError` from the `Error` family. With `ClassNotFoundException` the program itself asks for a class by name. With `NoClassDefFoundError`, a class that existed at compile time could not be found while the program was running; this is a deployment problem.

6.8.5 `NotSerializableException`

Definition 6.10

Serialization is the conversion of an object into a stream of bytes. This stream can be written to a file or sent over a network. Later the stream is converted back into an object.

In Java, for the objects of a class to be serializable, the class must implement the `Serializable` interface. This interface has no methods; it is only a permission marker. If an object of a class that does not implement the interface is to be written, a `NotSerializableException` occurs.

```
class Point {               // not Serializable
  int x = 3, y = 4;
}
// in main
try (ObjectOutputStream outputStream =
    new ObjectOutputStream(new ByteArrayOutputStream())) {
  // Try to serialize the object
  Point notSerializableObj = new Point();
  outputStream.writeObject(notSerializableObj);
} catch (NotSerializableException e) {
  System.out.println(e);
} catch (IOException e) {
  System.out.println("Other I/O error: " + e);
}
```

```
java.io.NotSerializableException: Point
```

`NotSerializableException` is an indirect subclass of `IOException`. For this reason its `catch` is written before that of `IOException`. If the order were reversed, the compiler would report `has already been caught`. The fix is to add the interface to the class:

```
class Point implements Serializable {
  int x = 3, y = 4;
}
```

With this fix the object can be written and read back. The test program wrote the object to a byte array and read it again, getting back the values `x = 3`, `y = 4`.

6.9 `Error` classes

`Error` classes represent system-level problems that fundamentally affect the normal running of the program. They are unchecked types. They generally arise not from a line of application code but from the state of the JVM and the runtime environment. The JVM itself throws them. Handling or declaring these errors is not recommended; letting the program terminate cleanly is a more suitable approach. Still, because they are `Throwable`s, they can technically be caught.

Error	When
`OutOfMemoryError`	Memory (heap) is exhausted

Error	When
`StackOverflowError`	The call stack overflowed; usually infinite recursion
`AssertionError`	An `assert` condition is false (only when `-ea` is on)
`NoClassDefFoundError`	A class that existed at compile time was not found at run time

Table 6.20: Commonly encountered `Error` classes.

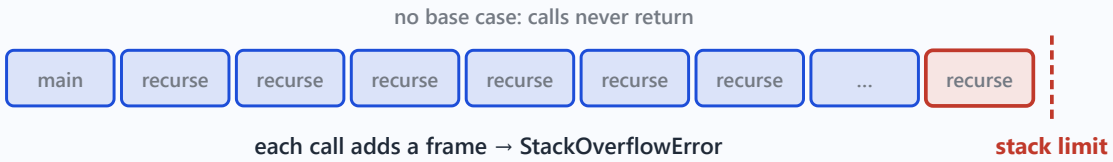


Figure 6.15: Infinite recursion fills the call stack. Because there is no base case, no call returns; each call adds a new frame, and when the stack limit is reached a `StackOverflowError` is thrown.

The program in the slides demonstrates all three errors. Here the only purpose of catching `Error`s is to see them on the screen.

```
static void recurse() {
  recurse();         // no base case
}
public static void main(String[] args) {
  try {
    recurse();
  } catch (StackOverflowError e) {
    System.out.println("SOE -> " + e);
  }
  try {
    long[] arr = new long[Integer.MAX_VALUE];
  } catch (OutOfMemoryError e) {
    System.out.println("OOM -> " + e);
  }
  int age = -5;
  assert age >= 0 : "Negative age: " + age;
  System.out.println("assert passed");
}
```

The program is run in two ways. With `java Demo` the output is:

```
SOE -> java.lang.StackOverflowError
OOM -> java.lang.OutOfMemoryError: Requested array size exceeds VM limit
assert passed
```

With `java -ea Demo` the last line changes:

```
SOE -> java.lang.StackOverflowError
OOM -> java.lang.OutOfMemoryError: Requested array size exceeds VM limit
Exception in thread "main" java.lang.AssertionError: Negative age: -5
    at Demo.main(Demo.java:17)
```

In the first `try`, the method `recurse()` calls itself endlessly. If the base case is forgotten in a recursion, each call adds a new frame and the stack fills up. In the second `try`, a `long` array of about two billion elements is requested. This array is larger than 16 GB; moreover, in the HotSpot JVM the maximum size of an array is slightly smaller than `Integer.MAX_VALUE`. The JVM rejects the request with the message `Requested array size exceeds VM limit`.

The third part is the `assert` statement. `assert condition : message;` throws an `AssertionError` with the message if the condition is false. But assertions (`assert`) are off by default; the JVM never runs them. That is why `assert passed` is printed on the first run. When they are turned on with the `-ea` (enable assertions) option, the condition is tested, an error is thrown because `age` is −5, and the program terminates. `assert` is for testing the programmer's assumptions during development; it is not used to check user input, because it can be switched off.

Example 6.9

The method `static int f(int n) { return f(n + 1); }` is called inside a `try` in `main`; `catch (Exception e)` prints `caught`; then `end` is printed. What happens when the program runs?

Neither `caught` nor `end` is printed. Each call adds a new frame to the stack; when the stack is full the JVM throws `StackOverflowError`. This is an `Error`, not in the `Exception` family. `catch (Exception e)` does not catch it; the program terminates with a long stack trace. In the listing the same `at` line is repeated hundreds of times. The compiler does not check for infinite recursion; the code compiles without trouble.

6.10 Custom exceptions

6.10.1 Writing a custom exception class

The exceptions in the Java library describe general situations. For the “insufficient balance” case in a banking program, or the “grade out of range” case in a grading system, the library has no special class. A developer can write custom exception classes to manage such situations that may arise in their own program.

Definition 6.11

A **custom exception class** is an exception class derived from `Exception` or `RuntimeException` that describes an application-specific situation.

```
public class MyException extends Exception {
  // Constructor
  public MyException(String message) {
    super(message);
  }
}
```

The constructor passes the message it receives up to the superclass with `super(message)`. The message is stored inside `Throwable`; so `getMessage()` and `toString()` show the message automatically. If the `super` call is forgotten, `getMessage()` returns `null`.

Superclass	Result
`extends Exception`	Checked: the caller must handle it; recoverable business-rule violations
`extends RuntimeException`	Unchecked: incorrect use, programmer error

Table 6.21: Choosing the superclass of a custom exception.

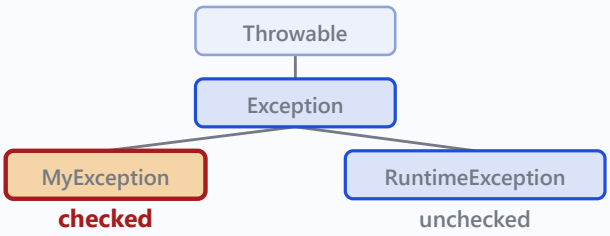


Figure 6.16: Where custom exception classes fit in the hierarchy. Because `MyException` extends `Exception`, it is a checked exception.

An object of any class that extends `Throwable` can be thrown. In practice, however, `Exception` or `RuntimeException` is extended. `Error` is reserved for JVM problems. Extending `Throwable` directly leaves the class outside both families; code that writes `catch (Exception e)` cannot catch it.

A few rules are followed when writing a custom exception. The name of the class describes the situation and ends with `Exception` (or `Error`): `InsufficientBalanceException`, `InvalidGradeException`. An exception is an ordinary class; it can contain extra fields and methods. Information such as the missing amount, the faulty value or the name of the faulty field can be added as fields. If the compiler is run with `-Xlint`, it also reports that the `serialVersionUID` field is missing; `Throwable` is a serializable class. This is a warning, not an error.

6.10.2 Throwing and catching a custom exception

```
public class MyExceptionTest {
  public static void main(String[] args) {
    try {
      doWork();
    } catch (MyException e) {
      System.out.println("Caught: " + e.getMessage());
    }
  }
  public static void doWork() throws MyException {
    if (true) {
      throw new MyException("Custom error occurred.");
    }
  }
}
```

```
Caught: Custom error occurred.
```

`MyException` is a checked exception because it derives from `Exception`. `throws MyException` is mandatory in the signature of `doWork()`. `main` uses the "handle" option by wrapping the call in a `try`.

The `if (true)` in the code is for demonstration only. Since `throw` is the last statement of the method here, writing it unconditionally would also compile. But if any code came after an unconditional `throw`, the compiler would report `unreachable statement`. In real programs the condition checks a business rule: is the balance sufficient, is the grade between 0 and 100, is the username empty?

Example 6.10

Let us write a checked `InvalidGradeException` class and a method `int enterGrade(int grade)`. If the grade is outside 0–100, the exception is thrown with the message `"Grade must be 0-100: …"`; if it is valid, the grade is returned. Let the grades 85, 120 and 40 be entered in a loop.

```
class InvalidGradeException extends Exception {
  InvalidGradeException(String m) { super(m); }
}
static int enterGrade(int grade)
    throws InvalidGradeException {
  if (grade < 0 || grade > 100) {
    throw new InvalidGradeException(
        "Grade must be 0-100: " + grade);
  }
  return grade;
}
```

```
for (int g : new int[]{85, 120, 40}) {
  try {
    System.out.println("Saved: "
        + enterGrade(g));
  } catch (InvalidGradeException e) {
    System.out.println("Error: "
        + e.getMessage());
  }
}
```

```
Saved: 85
Error: Grade must be 0-100: 120
Saved: 40
```

Because the class extends `Exception`, `throws` is mandatory. The `try` is inside the loop: the error for 120 affects only that iteration and does not prevent 40 from being saved. If the `try` were outside the loop, the loop would also end at the first error and 40 would never be tried.

6.10.3 An exception that carries information: a bank account

The real power of a custom exception is that it can carry information describing the situation. The following class stores the missing amount as well as the message.

```
class LowBalanceException extends Exception {
  private double deficit;

  public LowBalanceException(String msg, double deficit) {
    super(msg);            // kept in Throwable
    this.deficit = deficit;
  }

  public double getDeficit() {
    return deficit;
  }
}
```

The bank account that uses this class and the test code are as follows:

```
class Account {
  private double balance;
  Account(double balance) { this.balance = balance; }
  public void withdraw(double amount)
      throws LowBalanceException {
    if (amount > balance) {
      throw new LowBalanceException(
        "Insufficient funds", amount - balance);
    }
    balance -= amount;
    System.out.println("Balance: " + balance);
  }
}
// in main
Account acct = new Account(100);
try {
  acct.withdraw(30);
  acct.withdraw(90);
  acct.withdraw(10);
} catch (LowBalanceException e) {
  System.out.println(e.getMessage()
    + ", deficit: " + e.getDeficit());
}
```

```
Balance: 70.0
Insufficient funds, deficit: 20.0
```

Step	Line	Call stack	Variables	New output
1	15: `Account acct = new Account(100);`	main()	acct=→Account	—

Step	Line	Call stack	Variables	New output
2	6: `if (amount > balance) {`	main() → withdraw(30.0)	amount=30.0, balance=100.0	—
3	10: `balance -= amount;` 11: `System.out.println("Balance: " + balance);`	main() → withdraw(30.0)	amount=30.0, balance=70.0	`Balance: 70.0`
4	6: `if (amount > balance) {`	main() → withdraw(90.0)	amount=90.0, balance=70.0	—
5	7: `throw new LowBalanceException(` 8: `"Insufficient funds", amount - balance);`	main() → withdraw(90.0)	amount=90.0, balance=70.0	—
6	20: `} catch (LowBalanceException e) {`	main()	e=→LowBalanceException	—
7	21: `System.out.println(e.getMessage()` 22: `+ ", deficit: " + e.getDeficit());`	main()	e=→LowBalanceException	`Insufficient funds, deficit: 20.0`

Table 6.22: Trace of the bank account example.

In step 1 the `Account` object is created on the heap and `acct` refers to it; the balance is 100. In the call `withdraw(30)`, `30 > 100` is false; there is no exception. The balance becomes 70 and `Balance: 70.0` is printed. In step 4, `withdraw(90)` is called. This time `90 > 70` is true.

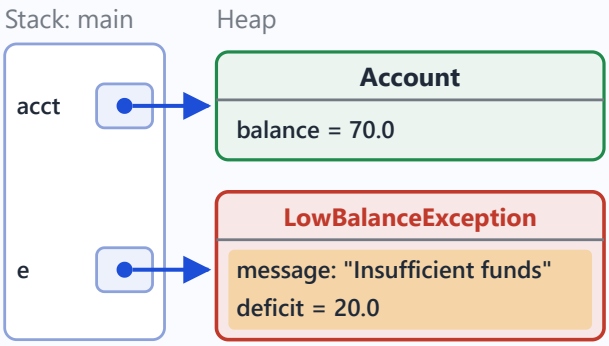


Figure 6.17: Step 7. There are two objects on the heap: `Account` and `LowBalanceException`. The `catch` parameter `e` refers to the thrown exception object. The balance stayed at 70.

In step 5 an exception object is created on the heap. The object carries two pieces of information: the message `"Insufficient funds"` and `deficit = 90 - 70 = 20.0`. The object is thrown. Because the line `balance -= amount` did not run, the balance does not change. In step 6 the frame of `withdraw` is discarded and `withdraw(10)` is skipped. The `catch` matches; `e` refers to the very object that was thrown. In step 7 the information in the object is used: `Insufficient funds, deficit: 20.0`.

This example makes it plain that an exception is also an ordinary object. It is created with `new`, it has fields, and its methods are called. The code that catches it can react meaningfully with this information. For example, it could ask the user, “You are 20 TL short; do you still want to try?” A return code such as `-1` could not have carried this much information.

A checked type was chosen for this class. Insufficient balance is a foreseeable situation and not a programmer's mistake. Every piece of code that wants to withdraw money must deal with this possibility; the compiler guarantees it.

Common mistake

Changing the state before checking the condition. If the line `balance -= amount;` were written before the `if`, the balance would already have dropped to −20 when the exception was thrown. The object would remain in a corrupted state. A good method does all the checks first, throws the exception if one is needed, and only then changes the state of the object.

6.11 Stack trace and good practices

6.11.1 What is a stack trace?

Definition 6.12

A **stack trace** is a report that shows which methods were on the call stack at the moment the exception occurred and where the exception occurred. The first line gives the type and message of the exception. Each following `at` line gives the class, method, file and line number of one call.

For an uncaught exception the JVM prints the stack trace to the error stream (`System.err`). It can also be printed at any time with `e.printStackTrace()`. The stack trace is the most important clue that helps the programmer understand and fix the error.

```
Exception in thread "main" java.lang.NullPointerException
    at com.example.MyClass.myMethod(MyClass.java:10)
    at com.example.MyClass.main(MyClass.java:6)
```

In this listing the `main` method called `myMethod` at line 6. The exception occurred at line 10 of `myMethod`. `com.example` is the name of the package and `MyClass` the name of the class. The list is read from top to bottom, from “most recently called” to “first called”.

Let us look at the method of reading on a more detailed example.

```
1  public class TraceDemo {
2    static int len(String text) {
3      return text.length();
4    }
5    static void show(String name) {
6      System.out.println(name + ": " + len(name));
7    }
8    public static void main(String[] args) {
9      show("Ali");
10     show(null);
11     System.out.println("Done");
12   }
13 }
```

```
Ali: 3
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()"
because "text" is null
    at TraceDemo.len(TraceDemo.java:3)
    at TraceDemo.show(TraceDemo.java:6)
    at TraceDemo.main(TraceDemo.java:10)
```

A stack trace is read in four steps:

1. **The first line** gives the type and message. Here the variable `text` is `null` and `length()` could not be called on it.
2. **The first `at` line** is where the exception was born: the method `len`, line 3.
3. **Downward** the call chain is read: `main` (10) → `show` (6) → `len` (3).
4. **Look first at the program's own lines.** Lines of library classes (those starting with `java.`) are mostly not the source of the problem.

The real cause is often further down the list. The exception was born at line 3, but there is nothing wrong at line 3: the method `len` correctly computes the length of the text it is given. It was line 10 that sent the `null` value. The place to fix is often not where the exception was born but where the faulty value was produced.

The name `"text"` in the message appears when the class is compiled with debugging information (`javac -g`). The same program compiled with plain `javac` cannot know the name and prints this:

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()"
because "<parameter1>" is null
```

Line numbers, however, are printed in both cases, because `javac` puts line information into the class file by default.

6.11.2 Code trace: where does the stack trace come from?

The lines of a stack trace are a snapshot: the moment the exception object is created, it records the call stack. The step-by-step trace in the slides shows how this snapshot is taken.

Step	Line	Call stack	Variables	New output
1	9: `show("Ali");`	main:9	—	—
2	6: `System.out.println(name + ": " + len(name));`	main:9 → show:6	name=Ali	—
3	3: `return text.length();`	main:9 → show:6 → len:3	text=Ali	—
4	6: `System.out.println(name + ": " + len(name));`	main:9 → show:6	name=Ali	`Ali: 3`
5	10: `show(null);`	main:10	—	—
6	6: `System.out.println(name + ": " + len(name));`	main:10 → show:6	name=null	—
7	3: `return text.length();`	main:10 → show:6 → len:3	text=null	—
8	3: `return text.length();`	main:10 → show:6 → len:3	text=null	—
9	10: `show(null);`	—	—	`Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()" because "text" is null ⏎ at TraceDemo.len(TraceDemo.java:3) ⏎ at TraceDemo.show(TraceDemo.java:6) ⏎ at TraceDemo.main(TraceDemo.java:10)`

Table 6.23: Trace of the `TraceDemo` program.

In step 1, `main` calls `show("Ali")` at line 9. `show` calls `len(name)` at line 6. Each frame remembers at which line its method is. `"Ali".length()` returns 3 and the frame is removed. `Ali: 3` is printed and `show` ends.

In step 5, `main` calls `show(null)` at line 10. `null` is passed as a parameter; at this point there is no problem yet. A reference being `null` is not an error; calling a method through `null` is. In step 7, `null.length()` is attempted and an NPE is thrown. At that moment there are three frames on the stack: `len:3`, `show:6`, `main:10`.

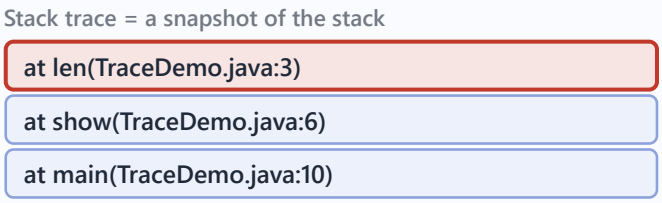


Figure 6.18: Step 8. The moment it is born, the exception object takes a snapshot of the stack. Each frame becomes an `at` line; the frame at the top of the stack is the first line of the list.

In step 8 the exception object records the stack. Each frame turns into an `at` line, and the topmost frame is written first. This is where the "reversed" order of the list comes from: the top of the stack is the head of the list. In the last step nobody has caught the exception. All three frames are discarded; the JVM prints the stack trace and terminates the program. `Done` is not printed.

Example 6.11

What does the following program print? According to the stack trace, at which line is the exception born, and where does the `null` value come from?

```
1 public class School {
2   public static void main(String[] args) {
3     int[] a = {70, 80};
4     int[] b = null;
5     report("A", a);
6     report("B", b);
7     System.out.println("done");
8   }
9   static void report(String name, int[] d) {
10    System.out.println(name + ": " + average(d));
11  }
12  static double average(int[] grades) {
13    int t = 0;
14    for (int n : grades) {
15      t += n;
16    }
17    return (double) t / grades.length;
18  }
19 }
```

```
A: 75.0
Exception in thread "main" java.lang.NullPointerException: Cannot read the array
length because "<local2>" is null
    at School.average(School.java:14)
    at School.report(School.java:10)
    at School.main(School.java:6)
```

The first `at` line shows where the exception was born: `average`, line 14. The for-each loop at this line tries to read the length of a `null` array. The message says so too: `Cannot read the array length`. `<local2>` appears instead of a variable name because the program was compiled without `-g`. The chain from bottom to top is `main` (6) → `report` (10) → `average` (14). `null` was assigned to `b` at line 4 and sent at line 6. Because nobody catches it, line 7 does not run.

6.11.3 Good practices in exception handling

Using the exception mechanism correctly is as important as knowing it. The following eight rules gather the lessons seen in the examples.

Catch a specific type
: If possible, catch the expected type instead of `catch (Exception e)` and give a reaction suited to each kind. The general type also silently swallows unexpected errors.

Do not swallow exceptions
: An empty `catch { }` hides the error. At the very least the exception is logged or reported to the user.

Close resources
: Resources such as files and connections are closed with try-with-resources or `finally`.

Prevent what can be prevented
: Do `null`, bounds and zero checks. Instead of catching a `RuntimeException`, fix the code.

Do not use them for flow control
: Ordinary tasks such as leaving a loop are not done with exceptions. Exceptions are for exceptional situations; also, creating an exception object and recording the stack is much more expensive than an ordinary `if`.

A meaningful message
: The message states the faulty value and the expectation: `"Grade must be 0-100: 120"`.

Handle in the right place
: The exception is caught where it can be resolved; if it cannot be resolved, it is passed on with `throws`.

Keep a log
: Logging exceptions to a file makes it easier to trace the error afterwards. A message that vanishes from the user's screen stays in the log file.

Common mistake

Swallowing the exception: `catch (Exception e) { }`. This code compiles and runs, but the error disappears without a trace. The program keeps running in a wrong state and the problem shows up much later in an unrelated place. By then the stack trace of the real cause is long gone.

6.11.4 `final`, `finally` and `finalize`

The three names are spelled alike, but their jobs are entirely different.

	What?	Purpose
`final`	Keyword	Variable: cannot be reassigned · method: cannot be overridden · class: cannot be subclassed
`finally`	Keyword (block)	A cleanup block after `try` that runs in every case, whether or not there is an exception
`finalize()`	A method of the `Object` class	The garbage collector used to be able to call it before deleting an object. When it will be called is undefined; it has been deprecated since Java 9.

Table 6.24: `final`, `finally` and `finalize()`.

The problem with `finalize()` is that it is unreliable. It is not clear when the garbage collector will delete an object, or even whether it will. It may never have run by the time the program ends. Since Java 18 this method has been marked for eventual removal; when a class that overrides it is compiled, `-Xlint` gives this warning: `finalize() in Object has been deprecated and marked for removal`. For resource cleanup, try-with-resources or `finally` is used, not `finalize()`.

6.12 Solved problems

The following problems are the example problems and exercises of the slides. Each solution is given step by step with its reasoning. All outputs were verified by running them with JDK 21.

Example 6.12 (Problem 1)

What does the following code print?

```
static int f() {
  try {
    System.out.print("A");
    int x = 5 / 0;
    System.out.print("B");
    return 1;
  } catch (ArithmeticException e) {
    System.out.print("C");
    return 2;
  } finally {
    System.out.print("D");
  }
}
// in main
System.out.println(" " + f());
```

Solution. To evaluate the argument of `println`, `main` first calls `f()`. `try` starts and `A` is printed. `5 / 0` throws an exception; `B` and `return 1` are skipped. The `catch` matches, prints `C` and prepares 2 with `return 2`. Before the method exits, `finally` runs and prints `D`. Then 2 is returned. `println` prints `" " + 2`. Result: `ACD 2`. Because `print` adds no line break, the four pieces are on the same line.

Example 6.13 (Problems 2 and 3)

```
try {
  String s = "12a";
  int n = Integer.parseInt(s);
  System.out.println("Number: " + n);
} catch (IllegalArgumentException e) {
  System.out.println("IAE");
} catch (Exception e) {
  System.out.println("Exception");
}
```

What does the code print? What happens if the two `catch` blocks are swapped?

Solution. `"12a"` is not a valid integer; `parseInt` throws a `NumberFormatException`. The first `catch` is for `IllegalArgumentException`. Because `NumberFormatException` is a subclass of this class, the first block matches: the output is `IAE`. The second block is not tried.

If the blocks are swapped, `catch (Exception e)` comes first. This block also covers `IllegalArgumentException`, and the second block becomes unreachable. The code does not compile:

```
Main.java:12: error: exception IllegalArgumentException has already been caught
} catch (IllegalArgumentException e) {
```

Example 6.14 (Problem 4)

```
static void a() {
  System.out.print("a1 ");
  b();
  System.out.print("a2 ");
}
static void b() {
  System.out.print("b1 ");
  throw new IllegalStateException("!");
}
// in main
try {
  a();
  System.out.print("m ");
} catch (IllegalStateException e) {
  System.out.print("caught" + e.getMessage());
} finally {
  System.out.println(" end");
}
```

The exception is born in `b()`; there is no `try` in `b()` or `a()`. Which `print`s run? When does `finally` run?

Solution. `main` calls `a()`: `a1` is printed. `a` calls `b()`: `b1` is printed. `b` throws an `IllegalStateException`. `b` has no `catch`; its frame is discarded. The exception passes to `a`; `a` has no `catch` either and `a2` is skipped. The exception reaches `main`; the rest of the `try` (`m`) is skipped. The `catch` matches and prints `caught!`; because the message is `"!"`, the two pieces are adjacent. Last of all, `finally` prints `end` and finishes the line. Result: `a1 b1 caught! end`.

Example 6.15 (Problem 5)

```
int[] d = {4, 0, 2};
int sum = 0;
for (int i = 0; i <= d.length; i++) {
  try {
    sum += 12 / d[i];
  } catch (ArithmeticException e) {
    sum += 100;
  } catch (ArrayIndexOutOfBoundsException e) {
    sum += 1000;
  }
}
System.out.println(sum);
```

What is `sum` ?

Solution. The loop condition is `i <= d.length` ; the loop runs for 0, 1, 2 and 3, that is, four iterations. Each iteration has its own `try` .

- `i = 0` : `12 / 4 = 3` ; sum 3.
- `i = 1` : `12 / 0` → `ArithmeticException` ; `+100` , sum 103.
- `i = 2` : `12 / 2 = 6` ; sum 109.
- `i = 3` : `d[3]` → `ArrayIndexOutOfBoundsException` ; `+1000` , sum 1109.

The output is `1109` . Because the `try` is inside the loop, an exception in one iteration does not end the loop. In the iteration `i = 1` , in the expression `sum += 12 / d[i]` the division is evaluated before the assignment; when the exception occurs, nothing is added to the sum.

Example 6.16 (Problem 6)

```
try {
  throw new RuntimeException("X");
} finally {
  System.out.println("F");
}
```

What does the code print?

Solution. The `try` throws a `RuntimeException` . There is no `catch` , but there is a `finally` . Before the exception leaves the method, `finally` runs and prints `F` . Then the exception continues on its way; because it is not caught in `main` either, the program terminates:

```
F
Exception in thread "main" java.lang.RuntimeException: X
    at P6.main(P6.java:4)
```

Example 6.17 (Exercise: reading an age)

Let us write a method `int readAge(String text)` that converts a string to an age. If the text is not a number, a `NumberFormatException` occurs on its own; if the number is negative, the method throws an `IllegalArgumentException`. Try the inputs `"20"`, `"-3"` and `"twenty"` with two `catch` blocks. What happens if the order of the `catch` blocks is reversed?

Solution. The method has three lines: `int age = Integer.parseInt(text);`, `if (age < 0) throw new IllegalArgumentException("Negative age: " + age);` and `return age;`. In the calling code, `catch (NumberFormatException e)` ("Not a number") is written first, then `catch (IllegalArgumentException e)`. The output: `Age: 20`, `Negative age: -3`, `Not a number: twenty`.

If the order is reversed, the `NumberFormatException` block can never be reached, because an NFE is an `IllegalArgumentException`. The compiler reports this error: `exception NumberFormatException has already been caught`.

Example 6.18 (Exercise: resources and the empty array)

How is code that closes a `BufferedReader` with `finally` written with try-with-resources? What should the method `double average(int[] grades)` do for an empty array?

Solution. The reader is opened in the parentheses: `try (BufferedReader r = new BufferedReader(new FileReader(path))) { … } catch (IOException e) { … }`. `close()` is called automatically; no `finally`, `null` check or nested `try` is needed.

The average of an empty array is undefined. Returning 0 is wrong, because 0 can be confused with a valid average; this is exactly the problem of the `-1` return code. An empty array is an invalid argument: `throw new IllegalArgumentException("Array is empty");`. An unchecked type is chosen, because passing an empty array is the caller's mistake. In the test program, `average(new int[0])` gave this exception, while the array `{70, 80, 90}` returned 80.0.

6.13 Common mistakes

Mistakes with exceptions fall into two groups. Compile-time errors mostly arise from the checked-exception rule and from the order of `catch` blocks; their messages are clear. Logic errors, on the other hand, pass the compiler and silently spoil the behavior of the program. The following two tables gather the mistakes treated separately in the body of the chapter. The compiler messages were obtained with the `javac` tool of JDK 21.

Mistake	javac message	Cause / fix
Forgetting a checked exception: `new FileReader(name)` with no `try` or `throws`	`unreported exception FileNotFoundException; must be caught or declared to be thrown`	Either `try`-`catch` or `throws`.
Writing the general `catch` first: `Exception` first, then `ArithmeticException`	`exception ArithmeticException has already been caught`	Subclass first, superclass after.
Catching what is never thrown: `catch (FileNotFoundException e)` on a `try` with no file operation	`exception FileNotFoundException is never thrown in body of corresponding try statement`	Remove the `catch`.
Writing only `try`	`'try' without 'catch', 'finally' or resource declarations`	At least one `catch` or `finally`.
Code after `throw` in the same block	`unreachable statement`	Remove the line, or make the `throw` conditional.
Using a variable from inside `try` outside it	`cannot find symbol`	Declare the variable before the `try`.
Throwing a string: `throw "error";`	`incompatible types: String cannot be converted to Throwable`	Throw an exception object.
`throws` in a body	`illegal start of expression`	`throw` in the body, `throws` in the signature.
Subclass and superclass in a multi-catch	`Alternatives in a multi-catch statement cannot be related by subclassing`	Write only the supertype.
A resource that is not `AutoCloseable`	`incompatible types: try-with-resources not applicable to variable type`	The class implements `AutoCloseable`.

Table 6.25: Common compile-time errors with exceptions.

Mistake	Consequence	Fix
Swallowing the exception: `catch (Exception e) { }`	The error disappears without a trace	At least log it or report it
Not checking a return code (`-1`)	The error shows up elsewhere, as a different exception	An exception instead of a return code
`return` inside `finally`	The pending exception and return value are silently erased	`finally` only does cleanup
Closing the resource at the end of the `try`	The resource stays open on an exception (leak)	try-with-resources

Mistake	Consequence	Fix
Believing that `catch (Exception e)` catches everything	`Error`s such as `StackOverflowError` are not caught	`Error`s are not caught; the cause is fixed
Catching an NPE and moving on	The source of the `null` is hidden	Find where the `null` comes from
Throwing an exception with no message	The message does not state the faulty value	Put the value and the expectation in the message
Using exceptions to control a loop	Hard-to-read and slow code	Ordinary cases are checked with `if`
Changing the state before the check	The object is left corrupted after an exception	Check first, then change

Table 6.26: Common logic errors with exceptions.

6.14 Summary

- An exception is an event that disrupts the normal flow while a program is running. In Java an exception is an object; it carries its type, its message and where it occurred. The JVM or the programmer (`throw`) can create it.
- Runtime problems have two branches: `Exception` if the program can recover, `Error` if there is a serious problem at the JVM or system level.
- A method in which an exception can occur either handles it (`try`-`catch`) or declares it to the caller (`throws`). The right place is where the exception can be resolved in a meaningful way.
- Special return codes (`-1`) mix the error with the normal result and slip by silently if unchecked. Exceptions solve these problems.
- If an exception occurs in `try`, the rest of the block is skipped and the first matching `catch` runs. After the `catch`, the program continues its normal flow. If there is no exception, the `catch` does not run.
- `catch` blocks are tried from top to bottom. A block catches the type written and all its subclasses. A subclass is written before its superclass; the reverse is the compile-time error `has already been caught`.
- Since Java 7, `catch (A | B e)` collects the same reaction in a single block. The alternatives cannot be subclasses of one another; `e` is implicitly `final`.
- `finally` runs in every case, even before a `return` inside `try` or `catch`. The return value is computed and stored first. `System.exit()` or a crash of the JVM prevents `finally`.
- try-with-resources closes the `AutoCloseable` resources opened in the parentheses automatically. Resources are opened in the order written and closed in reverse order, before the `catch`.
- In a nested `try`, an exception not caught at the inner level moves outward. The `finally` of every level it passes through runs.
- The root of the hierarchy is `Throwable`; its two subclasses are `Error` and `Exception`. `RuntimeException` is a subclass of `Exception`.

- For checked exceptions (those that derive from `Exception` but not from `RuntimeException`) the compiler enforces the "catch or declare" rule. The `RuntimeException` and `Error` families are not checked.
- `throw` throws an object in the body; `throws` declares in the signature the types that may be thrown.
- An uncaught exception propagates down the call stack toward the caller; the methods on the way end at that moment. This process is called stack unwinding. If `main` does not catch it either, the JVM prints the stack trace and terminates the program.
- Commonly seen runtime exceptions: `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException`, `IllegalArgumentException`, `NumberFormatException`, `ClassCastException`. These are mostly not caught but prevented.
- `double` operations do not throw exceptions (`Infinity`, `NaN`); an `int` addition that overflows silently wraps around. `Math.addExact` is used to check for overflow.
- Commonly seen checked exceptions: `IOException` and its subclass `FileNotFoundException`, `SQLException`, `ParseException`, `ClassNotFoundException`, `InterruptedException`.
- `Error`s (`OutOfMemoryError`, `StackOverflowError`, `AssertionError`) are not caught in application code. `assert` is turned on only with `-ea`.
- A custom exception class is written by extending `Exception` (checked) or `RuntimeException` (unchecked); the message is passed with `super(message)`. An exception can carry extra fields with information describing the situation.
- The first line of a stack trace is the type and message; the first `at` line is where the exception was born. The call chain is read downward. The real cause is often on the line that produced the faulty value.
- Good practices: catch a specific type, do not swallow the exception, close resources, and prevent what can be prevented. Exceptions are not used for flow control; the message is meaningful, the exception is handled in the right place and logged.

6.15 Key terms

exception runtime error logic error Error class throw / catch try-catch block finally block try-with-resources checked exception unchecked exception exception propagation call stack stack trace custom exception debugging assertion serialization garbage collector overflow cast (explicit conversion) generics method signature stack unwinding multi-catch resource leak pattern matching

6.16 Test Yourself

1. The outer `try` prints `A`. The inner `try` does `throw new RuntimeException("x")`; the inner `catch` catches only `IllegalStateException` and prints `B`; the inner `finally` prints `C`. After the inner block, the outer `try` prints `D`. The outer `catch (RuntimeException e)` prints `outer:` followed by the message. If all printing is done with `print`, what is the output?

   Answer

   `AC outer:x`. `RuntimeException` is the superclass of `IllegalStateException`, not a subclass; the inner `catch` does not match and `B` is not printed. Before the exception leaves, the inner `finally` prints `C`. `D` is skipped and the outer `catch` catches.

2. What does the method `static int f() { try { return 1; } finally { return 2; } }` return? Why is this style not recommended?

   Answer

   2. The `return 1` in the `try` prepares the value, but before the method exits `finally` runs and its own `return` replaces the pending value. The same style would also silently destroy an exception thrown in the `try`. This is why `javac -Xlint` gives the warning `finally clause cannot complete normally`.

3. What are the results of the calls `Integer.parseInt("12 ")` and `Double.parseDouble(" 12 ")`?

   Answer

   The first throws `NumberFormatException: For input string: "12 "`; `parseInt` does not accept trailing whitespace and `trim()` is needed first. The second returns `12.0`: `Double.parseDouble` strips leading and trailing whitespace itself. That two similar methods behave differently is one reason to always clean user input with `trim()`.

4. The parentheses of a try-with-resources contain `K a = new K("A"); K b = new K(null)`. The constructor of `K` throws `IllegalArgumentException("no name")` if the name is `null`, and otherwise prints `… opened`; `close()` prints `… closed`. The block prints `work`; the `catch` prints `catch` followed by the message. What is the output?

   Answer

   `A opened`, `A closed`, `catch no name`. Because the constructor of the second resource throws an exception, `b` is never created and the block body does not run. The successfully opened `a` is still closed. Only resources that could be created are closed.

5. Does `catch (IOException | Exception e)` compile? If the methods `static void a() throws IOException` and `static void b() throws RuntimeException` are called in `main` without a `try`, which call gives an error?

   Answer

   The multi-catch does not compile: `IOException` is a subclass of `Exception`, and the alternatives cannot be related by inheritance. The call `a()` gives the error `unreported exception IOException`. `b()` compiles: `RuntimeException` is unchecked; its declaration serves only as documentation.

6. What do `Integer.MAX_VALUE * 2` and `Math.multiplyExact(Integer.MAX_VALUE, 2)` give?

   Answer

   The first gives `-2` with no exception: the result exceeds the `int` range and the value wraps around (silent overflow). The second throws `ArithmeticException: integer overflow`. The exact methods make overflow visible.

7. For the class `class X extends Exception { X(String m) { } }`, what does `new X("m").getMessage()` return?

   Answer

   `null`. The constructor did not pass the message to the superclass with `super(m)`. The compiler implicitly calls the no-argument `super()`, and the message field of `Throwable` stays empty. The correct form is `X(String m) { super(m); }`.

8. A stack trace reads: `java.lang.ArithmeticException: / by zero`, followed by `at Account.ratio(Account.java:12)`, `at Account.report(Account.java:8)`, `at Account.main(Account.java:3)`. Where was the exception born? Which method did `main` call, and at which line?

   Answer

   The exception was born at the first `at` line, at line 12 of the `ratio` method. Reading the list from bottom to top, `main` called `report` at line 3, and `report` called `ratio` at line 8. The zero denominator was most likely produced in `report` or in `main`; to fix it, one looks back from there.

6.17 Further reading

- Oracle, *The Java Tutorials*, “Lesson: Exceptions”: the official and free treatment of the topic, with the pages on `try`, `catch`, `finally`, try-with-resources and “Unchecked Exceptions — The Controversy”.
- J. Bloch, *Effective Java* (3rd edition), Chapter 10, Items 69–77: discusses, with reasons, using exceptions only for exceptional conditions, choosing between checked and unchecked types, and not swallowing exceptions.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals* (12th edition), Chapter 7 “Exceptions, Assertions, and Logging”: treats exceptions together with `assert` and logging, with a practical orientation.
- J. Gosling et al., *The Java Language Specification, Java SE 21 Edition*, Chapter 11 “Exceptions” and 14.20 “The try statement”: the precise definition of the checked-exception rule and of `finally`.

6.18 References

1. Oracle, *The Java Tutorials*: Lesson: Exceptions (Catching and Handling Exceptions; The try-with-resources Statement; Unchecked Exceptions — The Controversy).
2. J. Gosling et al., *The Java Language Specification, Java SE 21 Edition*, Chapter 11: Exceptions; 14.20 The try statement.
3. Oracle, *Java SE 21 API Documentation*: `java.lang.Throwable`, `Exception`, `RuntimeException`, `Error`.
4. P. Deitel and H. Deitel, *Java: How to Program*, 11th edition, Pearson (Chapter 11: Exception Handling: A Deeper Look).
5. K. Sierra, B. Bates and T. Gee, *Head First Java*, 3rd edition, O'Reilly (Risky Behavior: exception handling).
6. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th edition, Pearson (Chapter 7: Exceptions, Assertions, and Logging).
7. J. Bloch, *Effective Java*, 3rd edition, Addison-Wesley (Chapter 10: Exceptions, Items 69–77).

Testing and Debugging

A program that compiles is not necessarily a program that works. The compiler checks only the syntax rules; it cannot know whether the program does what it was meant to do. That answer comes from testing. Testing systematically checks whether a program shows the expected behavior. When a test fails, debugging begins: the cause of the failure is found and fixed. This note first separates the notions of error, defect, and failure, and then introduces the test levels from unit to acceptance. At the center is automated unit testing with JUnit 5: assertion methods, `@BeforeEach`, exception tests, the test lifecycle, and parameterized tests. Boundary values, regression testing, and test-driven development complete the picture of good test design. The second half covers the debugger, breakpoints, stepping commands, and `jdb`. The note ends with profilers, code coverage, static analysis, property-based testing, Selenium, and memory analysis tools.

LEARNING OUTCOMES

Test levels
Distinguish unit, module, sub-system, system, and acceptance testing, and explain what each one verifies.

Unit testing with JUnit
Write automated tests with `@Test`, `@BeforeEach`, and assertion methods such as `assertEquals` and `assertThrows`; read the message of a failing test.

Good test design
Choose boundary values, write parameterized tests, keep tests deterministic and repeatable, and explain the value of regression testing.

Debugging
Set a breakpoint, step through the program (Step Into / Over / Out), and watch variables to find the source of a bug.

Analysis tools
Know which question each of these answers: profiler, code coverage, static analysis (SpotBugs), property-based testing, Selenium, Memory Analyzer, and VisualVM.

The right tool for the right question
Decide whether a given problem calls for a test, a debugger, or a profiler.

7.1 The foundations of testing and debugging

A student compiles a homework program, tries a few numbers, sees that the results look right, and hands it in. The instructor runs it with negative numbers, an empty array, or a very large input, and the program crashes or returns a wrong result. The student's quick try was also a test, but it was random and incomplete. This note is about making that try systematic, repeatable, and automatic.

7.1.1 Why test?

Many sayings circulate about testing in the software world. The software tester Michael Bolton stresses that the goal of testing is less to hunt for bugs than to make sure the system works as expected. An old joke among programmers calls a bug "an undiscovered feature." The saying that treats quality as a habit rather than an act is usually attributed to Aristotle; it is in fact Will Durant's reading of Aristotle. Shipping software without testing is also often compared to Russian roulette.

The most famous of these sayings belongs to Edsger W. Dijkstra. According to Dijkstra, testing can show the presence of bugs, but never their absence. This sentence draws the limit of testing clearly. However large a test suite is, it says something only about the inputs that were tried. One of the billions of untried inputs may crash the program.

Definition 7.1

Testing is the systematic check of whether a program shows the expected behavior. The program is given selected inputs, and the result it produces is compared with the expected result. If the results agree, the test *passes*; if not, it *fails*.

The words "expected result" in the definition are important. To write a test, one must first know what the program is supposed to do. "A method that adds two numbers should return 5 for 2 and 3" is an expectation; the test turns that expectation into code. A program whose expectation is unclear cannot be tested. For this reason, writing tests often helps to clarify a program's requirements as well.

No test suite proves that a program is free of defects, but well-chosen tests raise confidence considerably. If a method shows the expected behavior on normal inputs, on boundaries, and on invalid inputs, it is likely to work correctly on the remaining inputs too. The value of a test lies in producing this confidence cheaply and quickly.

7.1.2 Error, defect, failure

In everyday language, the word "bug" is used for everything: the programmer's misunderstanding, the wrong line in the code, and the wrong result on the screen are all called a bug. Software engineering gives these three things separate names. The distinction makes it easier to understand what testing and debugging each look for.

Definition 7.2

An **error** is a mistake in the programmer's mind: a wrong assumption, a forgotten case. A **defect** (or **bug**) is the wrong line of code in which that mistake shows up. A **failure** is the wrong behavior that appears when the defective code runs: wrong output, a crash, a freeze.

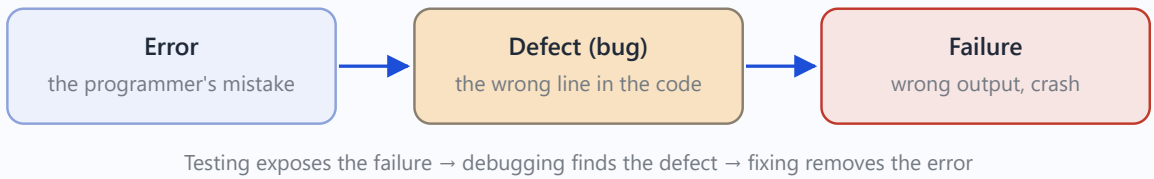


Figure 7.1: The fault chain. A human error enters the code as a defect; when the defective line runs, a failure is seen. Testing exposes the failure, debugging finds the defect, and the fix removes the error.

Figure 7.1 shows the chain from left to right. Each link of the chain lives in a different place. The error is in the programmer's head, the defect is in the source code, and the failure is in the behavior of the running program. Not every defect causes a failure. If the defective line never runs, or gives a wrong result only for rare inputs, the program may look fine for a long time.

Example 7.1

A programmer writes a method that finds the largest element of an array. The variable that holds the maximum is initialized to 0: `int best = 0;`. The three links in this program are as follows.

Error: the programmer assumed that "the array contains at least one positive number." For temperature readings, this assumption does not hold in winter.

Defect: the line `int best = 0;` in the code. If all elements are negative, no element can exceed 0.

Failure: for temperatures `{-5, -2, -8}`, the screen shows `Highest: 0`. The value 0 is not even in the array.

The same method gives the correct result (9) for `{3, 9, 4}`. The defect turns into a failure only when all elements are negative. For this reason, a try with positive numbers alone never exposes the defect.

7.1.3 Testing versus debugging

Testing and debugging are often confused. They serve the same goal but answer different questions. Testing asks "does the program work?" Debugging asks "why does it not work?" once a failure is known to exist.

Definition 7.3

Debugging is the process of finding and fixing the cause of a known failure, that is, the defective line. The tool used for this is called a **debugger**.

	Testing	Debugging
Goal	Check whether the program works as expected	Find and fix the cause of a known failure
Question	"Does it work?"	"Why does it not work?"
Result	Pass / fail (✓ / ✗)	The faulty line and its fix
Tools	JUnit, Selenium, code coverage	Debugger, stack trace, `println`
When	Continuously, after every change	When a test fails or a user reports a bug

Table 7.1: Comparison of testing and debugging.

Table 7.1 also shows how the two activities follow each other. Tests run continuously and report a failure. When a failure is seen, debugging steps in and finds the defect. After the defect is fixed, the tests run again: they confirm that the fix worked and that nothing else broke. A good software process repeats this cycle many times.

The "stack trace" in the table is the list of calls that the JVM prints when an exception is not caught. It shows which method threw the exception at which line and who called that method. In debugging, this is usually the first piece of information to look at.

7.2 Test levels

A piece of software consists not of a single method but of hundreds of interconnected classes. For this reason, testing is not done in a single step either. First the smallest parts are tested one by one, then the parts are tested working together, and finally the whole system is checked from the user's point of view. Each step is a different test level.

7.2.1 The levels of testing

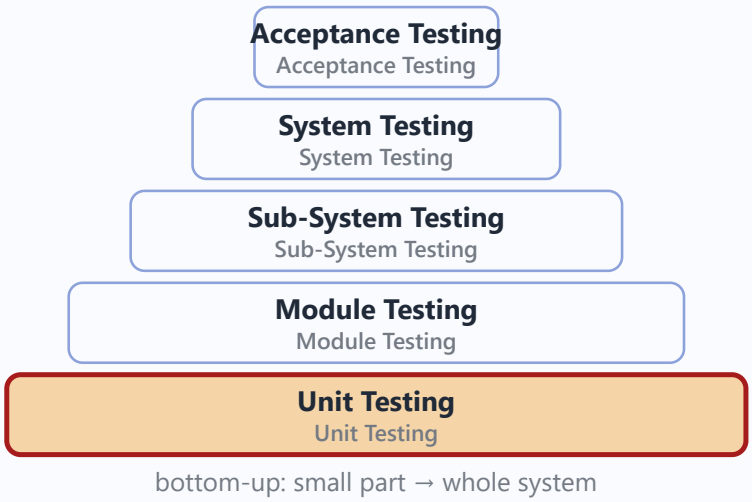


Figure 7.2: The test level pyramid. The base holds the most numerous, smallest, and fastest tests (unit testing); the top holds the fewest, largest, and slowest tests (acceptance testing).

Figure 7.2 shows the five levels as a pyramid. The shape of the pyramid is deliberate. The base is wide because unit tests are cheap and are written in large numbers. The top is narrow because acceptance tests are expensive, slow, and few. Going from the bottom to the top, the part under test grows.

Unit testing focuses on individual, independent components: a class or a method. It makes sure that each component works correctly on its own. For example, does the `withdraw()` method of a `BankAccount` class throw an exception when the balance is insufficient? Unit testing examines the smallest part; it is the most numerous and the fastest test.

Module testing examines a module made up of related components. It verifies that the components inside the module fit together and work together. For example, do the classes `Account`, `Transaction`, and `Receipt` carry out a money transfer correctly together? Even if every class is correct on its own, the cooperation among them may be wrong.

Sub-system testing looks for interface mismatches between sub-systems. It checks the communication and the data flow between components. For example, do the account sub-system and the database sub-system use the same date format? If one expects day/month/year and the other year-month-day, each sub-system works flawlessly alone, but together they produce a wrong record.

System testing examines the whole system as one. Both the functional requirements (what the system does) and the non-functional requirements (speed, security, usability) are checked. "Does a transfer complete in under 2 seconds with 1000 concurrent users?" is a system-testing question. No unit test can answer it.

Acceptance testing examines the system with real data instead of simulated data. It verifies the system's usability and performance in the real world. Acceptance testing is done together with the users. Alpha testing takes place in the developer's environment, beta testing in the user's own environment. In the end, a single question is answered: can the customer say "I accept this system"?

7.2.2 The levels at a glance

Level	What does it test?	Who does it?	Data
Unit	A single class / method	Developer	Small, hand-picked
Module	A few classes working together	Developer	Scenario data
Sub-system	Interfaces between sub-systems	Developer, test team	Scenario data
System	The whole system; speed, security	Test team	Close to real
Acceptance (alpha / beta)	Customer requirements	Together with the user	Real data

Table 7.2: Comparison of the test levels.

Table 7.2 also shows who owns each level. The developer writes the lower levels personally; at the higher levels a separate test team and finally the customer come into play. The data also gets closer to reality as one moves up.

The most important difference between the levels is cost. A bug at a low level is found cheaply and quickly. When a unit test fails, the location of the defect is almost known: a single method. If the same bug shows up in acceptance testing or in the user's hands, one first has to search for which part it comes from, then fix it, and then retest the whole system. The cost of the fix grows multiplicatively. For this reason, the aim is to catch bugs as low as possible.

Example 7.2

Which level catches each of the following problems most directly?

(a) The `calculateInterest()` method forgets to divide the percentage by 100. This is a logic error in a single method: *unit testing*.

(b) The `Account` class expects the amount in kuruş, while the `Receipt` class expects it in lira. This concerns the two classes working together: *module testing*.

(c) The payment sub-system sends dates as day/month/year, while the reporting sub-system sends them as year-month-day. This lies at the interface where the sub-systems meet: *sub-system testing*.

(d) The system stops responding with 500 users. This is a non-functional requirement: *system testing*.

(e) The customer says the report does not fit their accounting routine. This is a real user requirement: *acceptance testing*.

7.2.3 Black-box and white-box testing

Besides their levels, tests are also divided in two according to whether they look inside the code.

Definition 7.4

Black-box testing is testing done with only the input and the expected output, without looking inside the code. The program is seen from outside as a closed box. **White-box testing**, on the other hand, is done with knowledge of the code's structure; the aim is to execute every branch and every loop at least once.

Acceptance and system tests are mostly black-box tests. The customer does not read the code; they only check whether the system does what they want. Unit tests and code coverage, on the other hand, use the white-box view. If a method contains an `if`, a good unit test suite exercises both the true and the false branch of that `if`. The two views complement each other. Black-box testing starts from the requirements and asks "what should it do?" White-box testing starts from the code and asks "which path have I never tried?"

7.3 Unit testing and JUnit

The base of the test levels is unit testing, and it is also the test a programmer writes most often. In Java, unit tests are almost always written with the JUnit framework. To see why JUnit is needed, let us first look at what testing without it is like.

7.3.1 Testing by hand

The class to be tested is a simple calculator with two methods.

```
public class Calculator {
  public int add(int a, int b) {
    return a + b;
  }

  public int multiply(int a, int b) {
    return a * b;
  }
}
```

Someone new to programming writes a `main` method to try this class, calls the methods, and prints the results on the screen.

```
public class ManualTest {
  public static void main(String[] args) {
    Calculator calc = new Calculator();
    System.out.println("2 + 3 = " + calc.add(2, 3));
    System.out.println("2 * -3 = " + calc.multiply(2, -3));
    // Correct? We look at the screen and decide.
  }
}
```

```
2 + 3 = 5
2 * -3 = -6
```

The output looks right: 2 + 3 = 5 and 2 × (−3) = −6. But the decision "it looks right" is made not by the program but by the human looking at the screen. The comment line in the code admits as much. This method has four basic problems.

- The human eye decides whether the result is correct. The eye gets tired and makes mistakes; it is very easy to overlook the difference between 0.30000000000000004 and 0.3.
- When the code changes, everything has to be tried by hand again. Nobody does that every time.
- The test code stays inside `main` and is deleted when the work is done. The effort spent is lost.
- For a hundred methods, a hundred lines of output must be read. In a large project this is practically impossible.

The solution is simple: the expected value is also written into the code, and the comparison is left to the computer. The program decides by itself: "I expected 5, I got 5" or "I expected 5, I got 6." This is called an **automated unit test**.

Definition 7.5

A **unit test** is an automated test that checks a method or a class in isolation. The test calls the method with a known input, compares the result with the expected value, and makes the decision itself: pass or fail.

7.3.2 The JUnit framework

JUnit is the most widely used unit testing framework for Java. A framework is a ready-made library that makes a particular job easier and that itself calls the programmer's code. Tests are written as ordinary Java methods; JUnit finds them, runs them, and reports the results. The programmer does not write `main` and does not worry about who calls the test methods in what order.

In Java, a *unit* usually means a class. The tests check the methods of that class one by one. JUnit was designed basically for unit testing, but it is flexible. It can also be used for module testing in scenarios involving several classes, and for system testing in system-level interactions. IntelliJ IDEA, Eclipse, NetBeans, VS Code, and build tools such as Maven and Gradle run JUnit tests directly.

Version	How is a test written?
JUnit 3	Inheriting from the `TestCase` class; methods whose names start with `test`
JUnit 4	The `@Test` annotation, the `org.junit` package
JUnit 5	`@Test`, the `org.junit.jupiter.api` package; lambdas, parameterized tests

Table 7.3: Writing tests in the JUnit versions.

Table 7.3 summarizes the development of JUnit. JUnit 3 recognized tests by inheritance and a naming convention. JUnit 4 started to use annotations instead. An annotation is a note that starts with `@` and gives information to the compiler or to a tool; `@Override` is also an annotation. This note uses JUnit 5. The part of JUnit 5 in which tests are written is called *Jupiter*; the word `jupiter` in the package names comes from this. In a project that uses Gradle, JUnit 5 is added with the following line:

```
testImplementation("org.junit.jupiter:junit-jupiter:5.11.4")
```

In Maven and Gradle projects, files follow a fixed layout. Test classes go in the `src/test/java` folder, and the classes under test go in the `src/main/java` folder. This way the test code is compiled and run together with the program, but does not end up in the package that goes to the end user.

7.3.3 The first JUnit 5 test

Let us now turn the manual try written for `Calculator` into a JUnit test.

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

class CalculatorTest {

  @Test
  void addsTwoNumbers() {
    Calculator calc = new Calculator();
    assertEquals(5, calc.add(2, 3));
  }

  @Test
  void multipliesByNegative() {
    Calculator calc = new Calculator();
    assertEquals(-6, calc.multiply(2, -3));
  }
}
```

Let us go through the code line by line. The first line imports the `@Test` annotation. The `import static` in the second line imports the `static` method `assertEquals` of the `Assertions` class. This way one does not have to write `Assertions.assertEquals(…)` every time. The test class is named by adding `Test` to the name of the class under test: `CalculatorTest`. This is not a rule, but it is a common habit.

Every test method in the class is marked with `@Test`. This annotation tells JUnit "this method is a test." A test method is `void` and takes no parameters. No `main` is needed. In JUnit 5, the test class and its methods do not have to be `public`; package access is enough.

Each test method creates an object, calls a method, and checks the result with `assertEquals(expected, actual)`. The order of the parameters matters: the expected value comes first, then the actual result. If the values are equal, the method returns silently and the test passes (✓). If they are not equal, it throws `AssertionFailedError` and the test fails (✗). When these two tests are run, the report looks like this:

```
✓ addsTwoNumbers()
✓ multipliesByNegative()
2 tests · 2 passed · 0 failed
```

The names of the test methods read like sentences: `addsTwoNumbers`, `multipliesByNegative`. This is a deliberate choice. When a test fails, the name in the report tells you at once what broke. A name such as `test1` only says "something broke."

Common mistake

Forgetting the `import static` line. Without this line, the `assertEquals` call does not compile:

```
import org.junit.jupiter.api.Test;
class ImportTest {
  @Test void t() { int x = 5; assertEquals(5, x); }
}
```

```
ImportTest.java:3: error: cannot find symbol
  @Test void t() { int x = 5; assertEquals(5, x); }
                              ^
  symbol:   method assertEquals(int,int)
  location: class ImportTest
1 error
```

The compiler looks for a method named `assertEquals` in the test class and cannot find it. The fix is to add the line `import static org.junit.jupiter.api.Assertions.*;`.

7.3.4 Assertion methods

Definition 7.6

An **assertion** is a statement that compares an expected result with the actual one, or checks an assumption. In JUnit 5, the assertion methods are the `static` methods of the `org.junit.jupiter.api.Assertions` class. If an assertion fails, the method throws `AssertionFailedError` and the test fails.

`assertEquals` is the most frequently used assertion, but it is not the only one. Table 7.4 lists the most common assertion methods. All the examples in the last column of the table pass; each of them was tried by running it with JUnit 5.11.

Method	When does it pass?	Example (all pass)
`assertEquals(e, a)`	`e` and `a` are equal (`equals`)	`assertEquals(5, 2 + 3)`
`assertEquals(e, a, delta)`	For floating-point numbers, the difference is ≤ delta	`assertEquals(0.3, 0.1 + 0.2, 1e-9)`
`assertNotEquals(u, a)`	Not equal	`assertNotEquals(4, 2 + 3)`
`assertTrue` / `assertFalse`	The condition is true / false	`assertFalse("abc".isEmpty())`

Method	When does it pass?	Example (all pass)
`assertNull` / `assertNotNull`	The value is `null` / is not `null`	`assertNotNull("x")`
`assertSame(a, b)`	The same object (`==`)	`assertSame(s, s)`
`assertArrayEquals`	Arrays are equal element by element	`assertArrayEquals(new int[] {1, 2}, new int[] {1, 2})`
`assertThrows(T.class, code)`	The code throws an exception of that type	`assertThrows(ArithmeticException.class, () -> …)`
`assertAll(…)`	All assertions (every one is tried)	`assertAll(() -> …, () -> …)`
`fail("message")`	Never; fails the test on purpose	`fail("not written yet")`

Table 7.4: JUnit 5 assertion methods.

There is a detail behind every row of the table. `assertEquals` uses the `equals` method for objects; for this reason, two `String`s with the same contents are considered equal. `assertSame`, in contrast, compares with `==` and passes only for two references that point to the same object. For arrays, `assertEquals` is misleading. The `equals` method of arrays comes from `Object` and compares only references. Two separate arrays with the same contents are therefore not considered equal:

```
assertEquals(new int[] {1, 2}, new int[] {1, 2});   /* fails */
```

```
expected: [I@4a22f9e2<[1, 2]> but was: [I@3c419631<[1, 2]>
```

In the message, `[I@4a22f9e2` shows the type of the array (`[I` : an `int` array) and its identity code; the `[1, 2]` inside the square brackets is its contents. JUnit adds this detail because the text forms of the two values are the same. The contents are the same; the objects are different. To compare arrays element by element, use `assertArrayEquals`. The identity codes change on every run.

Floating-point numbers call for separate care. 0.1 and 0.2 cannot be represented exactly in binary; their sum is not 0.3 but 0.30000000000000004. For this reason, a tolerance is given as the third parameter for floating-point numbers. The call `assertEquals(0.3, 0.1 + 0.2, 1e-9)` passes because the difference between the two values is smaller than 10^{-9}. If the tolerance is forgotten, the test fails:

```
expected: <0.3> but was: <0.30000000000000004>
```

Most assertion methods also take an optional message as the last parameter. When a test fails, the message is added to the front of the report and explains what went wrong. The call `assertEquals(4, 2 + 3, "wrong sum")` fails with this message:

```
wrong sum ==> expected: <4> but was: <5>
```

`assertAll` groups several assertions together. When ordinary assertions are written one after another, the first failing assertion ends the test; the later ones never run. `assertAll`, on the other hand, runs all the assertions and reports all the failing ones together:

```
@Test
void twoAssertions() {
  assertAll(
    () -> assertEquals(4, 2 + 3),
    () -> assertEquals(7, 3 + 3));
}
```

```
org.opentest4j.MultipleFailuresError: Multiple Failures (2 failures)
        org.opentest4j.AssertionFailedError: expected: <4> but was: <5>
        org.opentest4j.AssertionFailedError: expected: <7> but was: <6>
```

Both assertions have failed, and the report shows both at once. When you check several fields of an object at the same time, this feature saves time: you see which fields are wrong in a single run. The parameters of `assertAll` are lambda expressions. A lambda expression is a small anonymous function written in the form `() -> …`; here it means "a piece of code to be run."

Common mistake

Writing the expected and the actual value the wrong way around. `assertEquals(calc.add(2, 3), 6)` also compiles and runs, but when it fails the message reads backwards: `expected: <5> but was: <6>`. The report seems to say "the method returned 6," although the method returned 5. Debugging with the wrong message goes to the wrong place. The rule: expected first, then actual.

7.3.5 The class under test: `BankAccount`

A realistic test example needs a somewhat richer class. The `BankAccount` class keeps a balance and checks its input in deposit and withdrawal operations.

```
public class BankAccount {
  private double balance;
  public BankAccount(double balance) {
    this.balance = balance;
  }
  public void deposit(double amount) {
    if (amount <= 0) {
      throw new IllegalArgumentException("Invalid amount");
    }
    balance += amount;
  }
  public void withdraw(double amount) {
    if (amount > balance) {
      throw new IllegalStateException("Insufficient funds");
    }
    balance -= amount;
  }
  public double getBalance() {
    return balance;
  }
}
```

The `deposit` method rejects a zero or negative amount with `IllegalArgumentException`. The `withdraw` method rejects an amount larger than the balance with `IllegalStateException`. The choice of the two exceptions is meaningful too. A negative amount is an invalid *argument*. Insufficient funds, on the other hand, means that it is not the argument but the object's current *state* that does not allow the operation.

Before writing tests, one thinks about which behaviors of the class are to be tested. Table 7.5 lists these behaviors for an account with an initial balance of 100.

Call	Expected	Kind
`deposit(50)`	balance 150	normal case
`withdraw(30)`	balance 70	normal case
`withdraw(500)`	`IllegalStateException`; balance stays 100	invalid input
`deposit(-10)`	`IllegalArgumentException`	invalid input
`deposit(0)`	`IllegalArgumentException`	boundary
`withdraw(100)`	balance 0, no exception	boundary

Table 7.5: Behaviors to be tested for `BankAccount` (initial balance 100).

Each row of the table becomes a separate test method. The rows fall into three groups: normal cases, invalid inputs, and boundaries. Testing only the normal cases is easy and very common; but bugs are mostly in the other two groups. The `withdraw(100)` row is a good example. If the condition had mistakenly been written as `amount >= balance` instead of `amount > balance`, withdrawing the entire balance would throw an exception. Only the input exactly on the boundary catches this defect.

7.3.6 The structure of a test: arrange, act, assert

A well-written test method has three parts. This pattern is known by the initials AAA (Arrange, Act, Assert).

```
class AaaTest {

  @Test
  void withdrawReducesBalance() {
    // Arrange
    BankAccount acct = new BankAccount(100);
    // Act
    acct.withdraw(30);
    // Assert
    assertEquals(70.0, acct.getBalance(), 0.001);
  }
}
```

In the **arrange** part, the objects and data needed for the test are set up: an account with a balance of 100. In the **act** part, the method under test is called once: `withdraw(30)`. In the **assert** part, the result is compared with the expected one: the balance should be 70. Because the balance is a `double`, a tolerance of 0.001 is given as the third parameter.

The AAA pattern makes a test readable. The reader sees at a glance what the test sets up, what it does, and what it expects. One consequence of the pattern is this: each test checks a single behavior. If a test has more than one "act" step, it is really several tests and should be split. This way, when a test fails, it is immediately clear which behavior broke.

7.3.7 Shared setup: `@BeforeEach`

The tests written for `BankAccount` all start with the same setup: an account with a balance of 100. Writing this line separately in every test is both repetition and a source of mistakes. JUnit offers the `@BeforeEach` annotation for shared setup.

```
class BankAccountTest {
  private BankAccount acct;

  @BeforeEach
  void setUp() {
    acct = new BankAccount(100);
  }

  @Test
  void depositIncreasesBalance() {
    acct.deposit(50);
    assertEquals(150.0, acct.getBalance(), 0.001);
  }

  @Test
  void overdraftThrows() {
    assertThrows(IllegalStateException.class,
      () -> acct.withdraw(500));
    assertEquals(100.0, acct.getBalance(), 0.001);
  }
}
```

`acct` is now a field of the test class. The `setUp` method, marked with `@BeforeEach`, runs before *every* test and assigns a new account to this field. This way every test starts with a fresh, clean account with a balance of 100.

There is a very important detail here. JUnit creates *a new object* of the test class for every test method. The `depositIncreasesBalance` test raises the balance to 150, but this change affects only the account in that test's object. The `overdraftThrows` test runs in its own test object with its own account. The tests do not affect each other.

The second test uses `assertThrows`. This method runs the code in its second parameter and expects that code to throw an exception of the type given in the first parameter. `() -> acct.withdraw(500)` is a lambda expression: it does not run immediately but is handed to `assertThrows` as "a piece of code to be run." `assertThrows` runs the code inside itself and catches the exception. Because the exception is caught, the test continues, and the second assertion checks that the balance has not changed. Seeing that an invalid operation did not corrupt the object is as important as seeing that the exception was thrown.

When these two tests are run, both pass:

```
✓ overdraftThrows()
✓ depositIncreasesBalance()
2 tests · 2 passed
```

The order in the report is not the order in the source code. By default, JUnit 5 orders the test methods with an algorithm that is deterministic but deliberately not specified. This order is the same on every run, but it does not have to match the order in the code.

7.3.8 Code trace: how does JUnit run tests?

Let us trace the execution of the `BankAccountTest` class step by step. At each step we will look at the objects on the heap: the test object and the `BankAccount` object it points to.

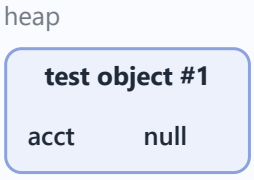


Figure 7.3: Step 1. JUnit finds the `@Test` methods in the class and creates a new test object for the first test. The `acct` field is still `null`.

In the first step, JUnit inspects the class with a mechanism called reflection and finds the methods marked with `@Test`. It then calls the no-argument constructor of the test class to create a test object. The `acct` field of this object holds its default value, `null` (Figure 7.3).

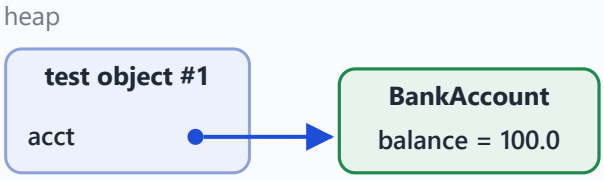


Figure 7.4: Step 2. `@BeforeEach` runs before the test: a new account is created, balance 100.

In the second step, `setUp` runs and the `acct` field points to a new `BankAccount` object. Next it is JUnit's turn: `overdraftThrows` runs first. `assertThrows` calls the lambda; `withdraw(500)` throws `IllegalStateException` because 500 > 100. The exception is of the expected type, so `assertThrows` passes. The balance is still 100; the second assertion passes as well (Figure 7.5).

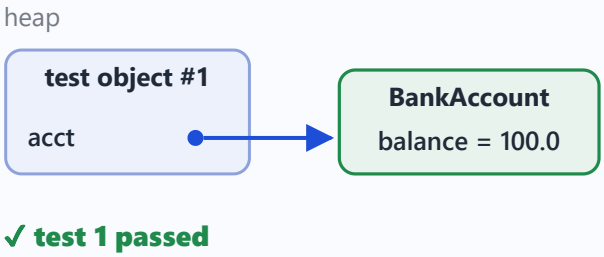


Figure 7.5: Step 4. The exception is of the expected type; `assertThrows` passes. The balance has stayed at 100, and the second assertion passes too.

For the second test, JUnit creates a brand-new test object (#2). The previous test object and its account are no longer used; the garbage collector will clean them up in due course. `@BeforeEach` runs again, and this test also starts with an account whose balance is 100. With the call `deposit(50)`, the balance becomes 150. In the assertion, 150.0 is compared with 150.0; the difference is less than 0.001, and the test passes (Figure 7.6).

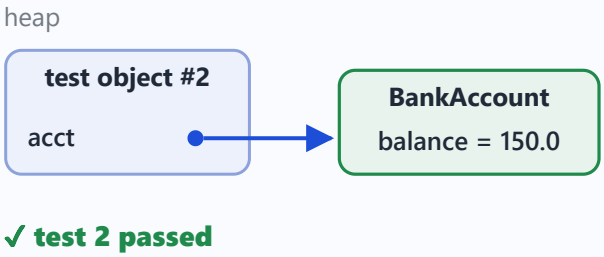


Figure 7.6: Step 8. The second test ran in its own test object (#2) with its own account. Both tests passed and remained independent of each other.

Step	Event	Test object	`acct`	Balance
1	Test object created	#1	`null`	—
2	`@BeforeEach`	#1	new account	100
3	`withdraw(500)`, exception	#1	same	100
4	both assertions pass ✓	#1	same	100
5	New test object	#2	`null`	—
6	`@BeforeEach`	#2	new account	100
7	`deposit(50)`	#2	same	150
8	assertion passes ✓	#2	same	150

Table 7.6: The state at each step while `BankAccountTest` runs.

Table 7.6 summarizes the trace. The most important row of the table is step 5: JUnit does not reuse the previous object for the second test. This design guarantees that tests are independent of each other. The state one test leaves behind does not leak into the next. This guarantee holds only for

instance fields; `static` fields belong to the class, and all test objects share the same field. If a test increments a `static` counter, the value seen by the next test changes as well.

7.3.9 Testing exceptions

Checking that invalid input is rejected is as important as checking valid input. A method that accepts a negative amount can be far more dangerous than one that gives a wrong result. `assertThrows` is made for this job.

```
class BankExceptionTest {

  @Test
  void overdraftMessage() {
    BankAccount acct = new BankAccount(100);
    IllegalStateException e = assertThrows(
      IllegalStateException.class, () -> acct.withdraw(500));
    assertEquals("Insufficient funds", e.getMessage());
  }

  @Test
  void negativeDepositRejected() {
    BankAccount acct = new BankAccount(100);
    assertThrows(IllegalArgumentException.class,
      () -> acct.deposit(-10));
  }

  @Test
  void wrongExpectation() {
    BankAccount acct = new BankAccount(100);
    assertThrows(IllegalStateException.class,
      () -> acct.withdraw(50));
  }
}
```

`assertThrows` does not merely check that an exception was thrown; it also returns the exception object that was thrown. The first test stores this object in the variable `e` and additionally verifies that its message is `"Insufficient funds"`. The second test checks that a negative deposit is rejected. The third test is deliberately wrong: withdrawing 50 TL is a valid operation and throws no exception. The result is as follows:

```
✓ overdraftMessage()
✓ negativeDepositRejected()
✗ wrongExpectation()
  Expected java.lang.IllegalStateException to be thrown, but nothing was thrown.
3 tests · 2 passed · 1 failed
```

The message is clear: `IllegalStateException` was expected, but no exception was thrown. `assertThrows` fails in two cases. The first is that no exception is thrown at all, as here. The second is that an exception of a different type than expected is thrown. For example, `assertThrows(IllegalStateException.class, () -> acct.deposit(-10))` fails with this message:

```
Unexpected exception type thrown, expected: <java.lang.IllegalStateException> but was:
<java.lang.IllegalArgumentException>
```

`assertThrows` accepts subtypes as well. `Integer.parseInt("abc")` throws a `NumberFormatException`; this class is a subclass of `IllegalArgumentException`. For this reason, `assertThrows(IllegalArgumentException.class, () -> Integer.parseInt("abc"))` passes. If the exact type is wanted, `assertThrowsExactly` is used; this method fails on the same call.

In JUnit 4, exception tests were written differently: `@Test(expected = IllegalStateException.class)`. With this form, the test passes if any line of the test method throws the exception. `assertThrows`, on the other hand, states exactly which call the exception is expected from and gives access to the exception object.

7.3.10 The test lifecycle

`@BeforeEach` is only one of JUnit's lifecycle annotations. Together, the four decide which code runs when around the tests.

```
class LifecycleTest {

  @BeforeAll
  static void start() { System.out.println("BeforeAll"); }

  @BeforeEach
  void setUp() { System.out.println("  BeforeEach"); }

  @Test
  void testA() { System.out.println("    testA"); }

  @Test
  void testB() { System.out.println("    testB"); }

  @AfterEach
  void cleanUp() { System.out.println("  AfterEach"); }

  @AfterAll
  static void finish() { System.out.println("AfterAll"); }
}
```

The only purpose of this class is to print the order in which the methods run. The output is as follows:

```
BeforeAll
  BeforeEach
    testA
  AfterEach
  BeforeEach
    testB
  AfterEach
AfterAll
```

Annotation	When does it run?	Typical use
`@BeforeAll`	Once before all tests (`static`)	Opening an expensive shared resource
`@BeforeEach`	Before every test	Rebuilding the test objects
`@AfterEach`	After every test (even if the test fails)	Deleting a temporary file
`@AfterAll`	Once after all tests (`static`)	Closing the shared resource

Table 7.7: JUnit 5 lifecycle annotations.

The output confirms the rules in Table 7.7. `BeforeAll` and `AfterAll` are printed once, at the very beginning and the very end. `BeforeEach` and `AfterEach` run once around each test; since there are two tests, they appear twice. It matters that `@AfterEach` runs even if the test fails: this is why cleanup work is put there. An exception thrown in the middle of a test does not leave a temporary file lying around.

The `@BeforeAll` and `@AfterAll` methods must be `static`. The reason can be seen in Table 7.6: JUnit creates a separate object for each test. There is no test object yet on which to call the method that is to run "once before all tests"; the method must belong to the class. If this rule is forgotten, JUnit runs none of the tests in the class and gives this error:

```
class NonStaticTest {
  @BeforeAll void start() { }
  @Test void t() { }
}
```

```
org.junit.platform.commons.JUnitException: @BeforeAll method 'void NonStaticTest.start()'
must be static unless the test class is annotated with @TestInstance(Lifecycle.PER_CLASS).
```

The `@TestInstance(Lifecycle.PER_CLASS)` annotation suggested by the message tells JUnit to use a single test object for all the tests. Then `@BeforeAll` can be an instance method; but the independence of the tests also weakens. The default behavior is usually safer.

The order of the tests among themselves is fixed, but it does not have to match the order in the source code. For this reason, tests are never written to depend on one another. An arrangement such as "first `testA` opens the account, then `testB` withdraws money from that account" breaks when the order

changes.

7.3.11 From red to green: a failing test

The real value of a test shows when it catches a bug. The first version of the `max` method from the fault-chain example is this:

```
public class ArrayUtils {
  public static int max(int[] arr) {
    int best = 0;
    for (int i = 0; i < arr.length; i++) {
      if (arr[i] > best) {
        best = arr[i];
      }
    }
    return best;
  }
}
```

The code compiles and looks right at first glance. The loop compares every element with `best` and updates `best` when it finds a larger one. Let us write two tests for this method: one with an array of positive numbers and one with an array of all negative numbers.

```
class ArrayUtilsTest {

  @Test
  void positiveArray() {
    int[] arr = {3, 9, 4};
    assertEquals(9, ArrayUtils.max(arr));
  }

  @Test
  void allNegativeArray() {
    int[] arr = {-5, -2, -8};
    assertEquals(-2, ArrayUtils.max(arr));
  }
}
```

```
✓ positiveArray()
✗ allNegativeArray()
  org.opentest4j.AssertionFailedError: expected: <-2> but was: <0>
2 tests · 1 passed · 1 failed
```

The message of the failing test has two parts. `expected` shows the expected value (−2), and `but was` shows the value the method actually returned (0). The value 0 is not even in the array. This clue almost gives away the location of the defect: the value 0 did not come from inside the array but from somewhere else. The only candidate is the initialization `best = 0`. When all elements are negative, no element can exceed 0, and `best` is never updated.

In the testing world, a failing test is called *red* and a passing test *green*; IDEs show the results in these colors. The corrected version of the defect is this:

```
public class ArrayUtils {
  public static int max(int[] arr) {
    int best = arr[0];               // first element
    for (int i = 1; i < arr.length; i++) {
      if (arr[i] > best) {
        best = arr[i];
      }
    }
    return best;
  }
}
```

The maximum is now initialized with the first element of the array. The loop also starts at index 1, because element 0 is already in `best`. The same two tests are run again, and both pass:

```
✓ positiveArray()
✓ allNegativeArray()
2 tests · 2 passed
```

`i`	`arr[i]`	**Version 1: condition**	**Version 1:** `best`	**Fixed: condition**	**Fixed:** `best`
start	—	—	0	—	−5
0	−5	−5 > 0 false	0	(loop starts at 1)	−5
1	−2	−2 > 0 false	0	−2 > −5 true	−2
2	−8	−8 > 0 false	0	−8 > −2 false	−2
return			**0 ✗**		**−2 ✓**

Table 7.8: Trace of the two versions for the array `{-5, -2, -8}`.

Table 7.8 shows the difference between the two versions step by step. In the first version, the condition is never true in any iteration. In the fixed version, in the first iteration (`i = 1`) −2 turns out to be greater than −5, and `best` is updated. The loop runs as many times as the length of the array in the first version and one fewer time in the fixed version; the time cost is O(n) and the extra memory is O(1).

This test has one more value: it is now permanent. If the code is changed again someday and the same defect comes back, the `allNegativeArray` test catches it immediately. That the same tests are run again after the fix and turn green is the evidence that the fix worked.

> **Common mistake**
>
> The fixed version raises a question for the empty array as well. `arr[0]` throws `ArrayIndexOutOfBoundsException` on an empty array. The first version would silently return 0 for the same input, which is also a wrong answer, because an empty array has no largest element. What a method does on empty input must be decided explicitly, and that decision must be tested too, for example `assertThrows(ArrayIndexOutOfBoundsException.class, () -> ArrayUtils.max(new int[0]))`.

7.4 Good test design

JUnit makes writing tests easier, but it does not decide which tests to write. The input space of a method is often huge: even a single parameter of type `int` can take more than four billion values. Trying them all is impossible. Good test design is the art of catching bugs with a small number of well-chosen inputs.

7.4.1 Boundary values and equivalence classes

The example method converts a score between 0 and 100 into a letter grade.

```
public class Grade {
  public static String letterGrade(int score) {
    if (score < 0 || score > 100) {
      throw new IllegalArgumentException("Invalid score: " + score);
    }
    if (score >= 90) return "AA";
    if (score >= 80) return "BA";
    if (score >= 70) return "BB";
    if (score >= 60) return "CB";
    if (score >= 50) return "CC";
    return "FF";
  }
}
```

The method first rejects out-of-range scores and then tries the thresholds from top to bottom. Which scores should be tested? There are 101 different valid scores; writing all of them is both tedious and unnecessary. The scores 61, 65, and 68 follow the same path and give the same result (CB). If one is right, the others are expected to be right as well.

Definition 7.7

An **equivalence class** is a set of inputs that lead to the same behavior in the program. Picking a single representative from an equivalence class is enough. **Boundary value analysis**, in turn, chooses the end values of each equivalence class and the first values of the invalid region as test inputs.

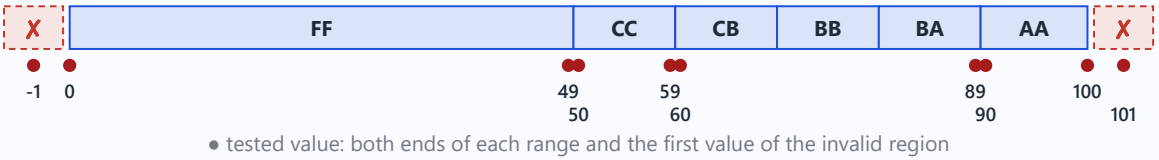


Figure 7.7: The score line from 0 to 100. The ranges FF, CC, CB, BB, BA, and AA; the marked points are the boundary values that are tested. The ✗ regions at the two ends are invalid scores.

Why boundary values matter so much is explained by one observation: bugs occur most often at the boundaries. Programmers often mix up `>` and `>=`, or `< 100` and `<= 100`. A defect of this kind never shows up for inputs in the middle of a range. A score of 65 gives CB with `score >= 60` and with `score > 60` alike. The difference appears only exactly on the boundary, at 60.

Figure 7.7 shows the chosen values. Both ends of each range are tested: 49 and 50 enclose the boundary between FF and CC from both sides; 89 and 90 do the same for BA and AA. The first values of the invalid region (−1 and 101) are chosen too, because a defect in the range check is seen only with them. The ends of the valid region (0 and 100) are not forgotten either.

Example 7.3

How is a complete set of boundary values built for `letterGrade`?

Equivalence classes: invalid low (< 0), FF (0–49), CC (50–59), CB (60–69), BB (70–79), BA (80–89), AA (90–100), invalid high (> 100). There are eight classes in total.

Take both ends of each valid class: 0, 49, 50, 59, 60, 69, 70, 79, 80, 89, 90, 100. Add the first values of the invalid region: −1 and 101. That makes 14 inputs.

This set tries only 12 of the 101 possible valid scores, but it tests every threshold from both sides. If any `>=` had mistakenly been written as `>`, the lower end of that threshold would give the wrong letter and the test would fail.

7.4.2 Parameterized tests

Once the boundary values are chosen, is a separate test method needed for each one? Writing an almost identical method for every value is both long and prone to copy-and-paste mistakes. JUnit 5 offers the parameterized test instead.

```
class GradeTest {

  @ParameterizedTest
  @CsvSource({"0, FF", "49, FF", "50, CC", "59, CC",
        "60, CB", "89, BA", "90, AA", "100, AA"})
  void boundaries(int score, String expected) {
    assertEquals(expected, Grade.letterGrade(score));
  }

  @Test
  void outOfRangeRejected() {
    assertThrows(IllegalArgumentException.class,
      () -> Grade.letterGrade(-1));
    assertThrows(IllegalArgumentException.class,
      () -> Grade.letterGrade(101));
  }
}
```

The `@ParameterizedTest` annotation states that the same test method will run again and again with different data. The data is supplied with `@CsvSource`. Each string is one test invocation. The comma-separated values are converted in order into the parameters of the method: in the string `"50, CC"`, `"50"` is converted to an `int` and passed to `score`, and `"CC"` is passed to `expected`. The name comes from the CSV (comma-separated values) file format.

Parameterized tests live in the `junit-jupiter-params` module. The `junit-jupiter` package added to Gradle includes this module as well; nothing extra needs to be added. When the class is run, the report shows each data row as a separate test:

```
boundaries(int, String)
  ✓ [1] 0, FF
  ✓ [2] 49, FF
  ✓ [3] 50, CC
  ✓ [4] 59, CC
  ✓ [5] 60, CB
  ✓ [6] 89, BA
  ✓ [7] 90, AA
  ✓ [8] 100, AA
✓ outOfRangeRejected()
9 tests · 9 passed
```

The report has 9 tests: the 8 invocations of the parameterized test and the ordinary test that checks the out-of-range scores. Had a row failed, the sequence number and the values in the square brackets would immediately show which data caused the problem. The out-of-range test calls `assertThrows` twice: once for −1 and once for 101.

Compared with the complete set in Example 7.3, this test skips a few boundaries: the pairs 69/70 (CB/BB) and 79/80 (BB/BA) are not in the list. The BB range is never tested. Adding four more strings to `@CsvSource` closes this gap; the test method itself does not change at all. This is the real strength of the parameterized test: adding a new case is a one-line job.

7.4.3 Code trace: a parameterized test

Let us trace the parameterized test along the score axis. JUnit first reads the 8 strings in `@CsvSource`; each of them will be a separate test invocation.



Figure 7.8: Row 1. `"0, FF"` is split at the comma; `"0"` is converted to an `int` and passed to `score`, and `"FF"` is passed to `expected`. The result is `"FF"`, and the test passes.

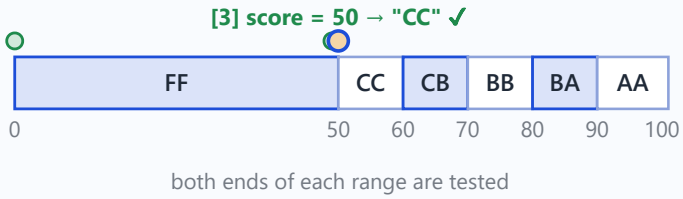


Figure 7.9: Row 3. 50 is the lower end of the CC range; together with 49, it tests the FF/CC boundary from both sides.

Call	`score`	`expected`	First true condition	Returned	Role	Result
[1]	0	FF	none	FF	first value of the valid region	✓
[2]	49	FF	none	FF	upper end of FF	✓
[3]	50	CC	`score >= 50`	CC	lower end of CC	✓
[4]	59	CC	`score >= 50`	CC	upper end of CC	✓
[5]	60	CB	`score >= 60`	CB	lower end of CB	✓
[6]	89	BA	`score >= 80`	BA	upper end of BA	✓
[7]	90	AA	`score >= 90`	AA	lower end of AA	✓

Call	`score`	`expected`	First true condition	Returned	Role	Result
[8]	100	AA	`score >= 90`	AA	last value of the valid region	✓

Table 7.9: The eight invocations of the parameterized test and the first condition that is true inside `letterGrade`.

Table 7.9 shows which path each call takes inside the method. The row for 49 is interesting: none of the conditions is true, and the method reaches the `return "FF"` at the bottom. If the condition had mistakenly been written as `score > 48` instead of `score >= 50`, 49 would give CC and [2] would fail. The row for 100 proves that the `score > 100` check is written correctly. If the check had mistakenly been `score >= 100`, a score of 100 would throw an exception.

Example 7.4

Suppose `score > 50` is mistakenly written instead of `score >= 50` in `letterGrade`. Which row of the parameterized test fails?

49: no condition is true, FF. Expected FF; passes.

50: `50 > 50` is false; the method returns FF. Expected CC; **fails**: `expected: <CC> but was: <FF>`.

59: `59 > 50` is true, CC. Expected CC; passes.

Only [3] fails. 49 and 59 cannot see this defect; only the value exactly on the boundary catches the bug.

7.4.4 Properties of a good unit test: FIRST

The properties of a good unit test are summarized by the acronym FIRST, formed from their initials.

F · Fast
: It finishes within milliseconds; thousands of them can be run after every change. A unit test does not use files, the network, or a database. If a test suite takes five minutes, programmers stop running it.

I · Independent
: It does not depend on another test, on the test order, or on the state left behind by a previous test. JUnit's creation of a new object for every test supports this property.

R · Repeatable
: It gives the same result on every machine at any time. Variable sources such as randomness and the clock are brought under control.

S · Self-validating
: The result is ✓ or ✗. No human needs to read and interpret the output.

T · Timely

> It is written together with the code, or even before the code. A test put off until months later is usually never written.

One more property is added to these five: readability. A descriptive name, the AAA pattern, and one behavior per test make a test readable. A well-written test is a document that shows how the code is to be used. A new programmer learns what the `withdraw` method does on insufficient funds by reading `BankAccountTest`.

7.4.5 Unit test or integration test?

Real classes often do not work alone. An order class reads prices from a database, writes the result to a file, and sends an e-mail. A test that checks such a class together with the real database is no longer a unit test.

	Unit test	Integration test
Scope	A single class / method, in isolation	Several components working together
Speed	Milliseconds	Seconds (database, file, network)
External dependency	Replaced by a simple fake object (test double)	Real database, file, service
Bugs it finds	Local logic errors; the location is immediately clear	Mismatched interfaces, wrong configuration

Table 7.10: Comparison of unit tests and integration tests.

An **integration test** checks components that work together with their real dependencies. Module and sub-system tests are of this kind. In a unit test, on the other hand, an external dependency is replaced by a simple object that imitates it. This object is called a **test double**; the name comes from the stunt double who takes the actor's place in dangerous scenes. For example, in place of an interface that reads prices from a database, a small class is written that returns a fixed price for every product. The test becomes both faster and independent of the current contents of the database.

Table 7.10 also shows that the two kinds of test do not replace each other. Unit tests find logic errors quickly and show their location exactly. Integration tests verify that the parts are wired together correctly. As in the test pyramid, unit tests are written in large numbers and integration tests in smaller numbers.

Test names should also describe behavior. Of the three names below, the first says nothing; the other two explain which rule was broken when the test fails.

```
@Test void test1() { … }
@Test void discountForSixItems() { … }
@Test void noDiscountForFive() { … }
```

In a unit test, what is checked is the externally visible behavior, not the internal details of the implementation. What matters about a method is what it returns, not which helper method it calls inside. Tests tied to internal details break at every refactoring, even when the behavior has not changed. A unit test does not use `Thread.sleep`, the real network, or files; otherwise the test becomes slow and starts to behave randomly. Mutable state, such as a `static` list, is not shared between tests either.

7.5 Regression and automation

The value of a test does not end on the day it is written. As the code changes, the test runs again and checks whether the old behavior is preserved. What makes tests so valuable is that they are automatic and can be rerun after every change.

7.5.1 Regression testing

> **Definition 7.8**
>
> **Regression testing** is verifying, after changes are made, that everything that worked before still works. Regression means "going backwards": something that had been fixed breaks again. The method is simple: all previously written tests are rerun after every change.

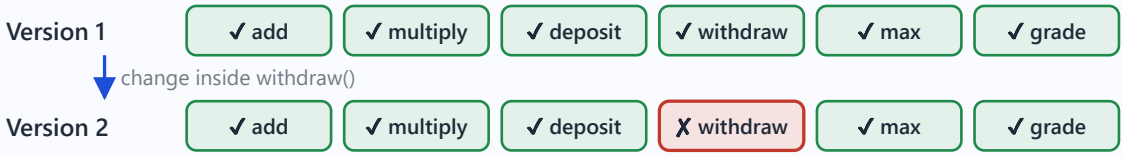


Figure 7.10: Six tests pass in version 1. After a change, the `withdraw` test fails in version 2: the change has broken an old behavior.

Figure 7.10 shows a typical regression. While adding a new feature, the programmer touched the `withdraw` method and broke it without noticing. Without tests, this defect would be noticed only when a customer could not withdraw money. Regression testing catches the defect right after the change, while the person who made it still has that code in mind.

For regression testing to work, tests must have two properties.

- **Deterministic**: it must give the same result every time under the same conditions. A test that sometimes passes and sometimes fails is called *flaky*. A flaky test destroys trust: when it fails, people say "that test again" and move on, and a real bug is overlooked too.

- **Repeatable**: it must be possible to repeat it in the same way at different times, on different machines, and in different situations. A test that passes only on the programmer's computer says nothing when it fails on the server.

7.5.2 Testing code that involves randomness deterministically

How is code that generates random numbers tested? A die class returns a different number between 1 and 6 on every roll; writing the expected result in advance seems impossible. The solution is to move the source of randomness outside the class.

```
import java.util.Random;

public class Die {
  private final Random random;

  public Die(Random random) {       // injected
    this.random = random;
  }

  public int roll() {
    return random.nextInt(6) + 1;
  }
}
```

The `Die` class does not create its own `Random` object; it receives it through its constructor. This design is called **dependency injection**: another object that an object needs is supplied from outside. In real use, the program writes `new Die(new Random())` and gets different numbers each time. The test, on the other hand, supplies a `Random` created with a fixed seed.

The `Random` class does not really produce random numbers but pseudo-random ones. The numbers are derived from a starting value by a formula. This starting value is called the **seed**. Two `Random` objects created with the same seed produce the same sequence of numbers. Five rolls made with seed 42 give the following result on every machine and in every run:

```
3 4 1 3 1
```

```
class DieTest {

  @Test
  void sameSeedSameRolls() {
    Die d1 = new Die(new Random(42));
    Die d2 = new Die(new Random(42));
    for (int i = 0; i < 100; i++) {
      assertEquals(d1.roll(), d2.roll());
    }
  }
}
```

The test creates two dice with the same seed and verifies that over 100 rolls the two always give the same number. The test gives the same result in every run; that is, it is deterministic. Had `new Random()` been written inside the die class, the test would have had no way at all of controlling the seed.

Determinism makes other tests possible too. `assertEquals(3, new Die(new Random(42)).roll())` always passes, because the first roll of seed 42 is always 3. Had the same assertion been written with an unseeded `new Random()`, the test would pass only with probability 1/6: a textbook flaky test. A meaningful test that can be written with an unseeded die checks the range of the result: an assertion such as `int x = die.roll(); assertTrue(x >= 1 && x <= 6);` holds for every die.

Common mistake

Testing clock-dependent code directly. A "birthday discount" method that calls `LocalDate.now()` gives a different result depending on the day the test is run. The test passes on the day it is written and fails the next day. The solution is the same as in the die example: the source of the date or time is supplied from outside, and the test uses a fixed date.

7.5.3 Automated tests and continuous integration

Code should be checked with automated tests instead of by hand. The effort spent is about the same: for a manual try, too, inputs are chosen and outputs are examined. The difference shows up in the end. Nothing remains from a manual try; from an automated one, a test remains that can be rerun whenever you like. This test can run again on every build and every change.

It is wrong to think that software need not be tested again just because it worked once. Code changes, the libraries in use move to new versions, and the environment the program runs in changes. A program that worked correctly yesterday can break today.

	Manual	Automated
Initial cost	Low	Somewhat higher
Cost of repetition	The same every time	≈ 0

	Manual	Automated
Speed	Minutes	Seconds
Reliability	Human error	The same every time
Regression	Not done in practice	On every change
Documentation value	None	Usage examples

Table 7.11: Comparison of manual and automated testing.

Table 7.11 compares the two approaches. The initial cost of an automated test is somewhat higher: the test code has to be written. But the cost of each repetition is almost zero. If a test is going to be run 100 times, the automated one is far cheaper. Regression testing can in practice be done only with automated tests; no programmer tries a hundred cases by hand after every change.

The last step of automation is for tests to run without anyone asking the programmer. **Continuous integration** (CI) systems do this. GitHub Actions and Jenkins are common examples of such systems.

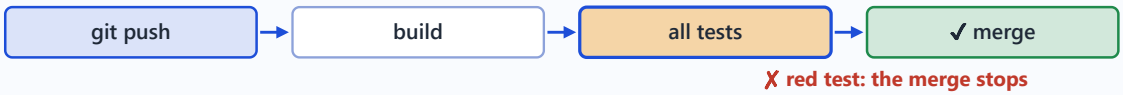


Figure 7.11: A continuous integration pipeline. On every push, the code is built on the server, the tests run, and the result is reported. A red test stops the merge.

Figure 7.11 shows a typical pipeline. The programmer pushes the change to the shared repository (`git push`). The CI server fetches the code, builds it, and runs all the tests. If one of the tests fails, the change is not merged into the main code and the programmer is notified. In this way no regression can enter the main code unnoticed.

7.5.4 Test-driven development

Tests are usually written after the code. Some programmers reverse the order: first the test, then the code.

Definition 7.9

Test-driven development (TDD) is a development method that writes the test before the code. The method proceeds in short cycles; each cycle has three steps: red, green, refactor.

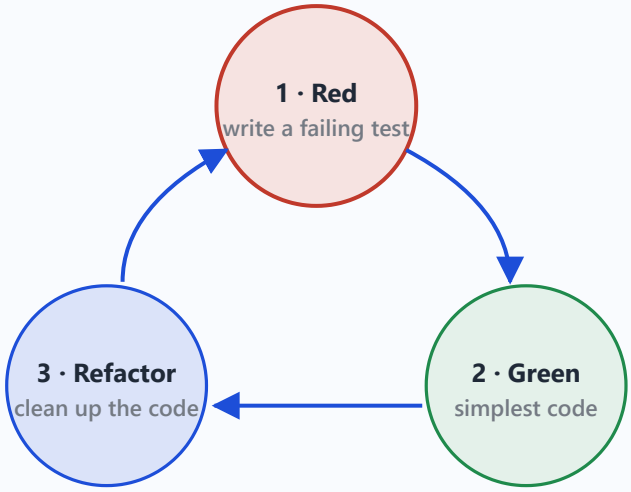


Figure 7.12: The TDD cycle. 1: a failing test is written (red). 2: the simplest code that makes the test pass is written (green). 3: the code is cleaned up while the tests stay green (refactoring).

Figure 7.12 shows the cycle. In the **red** step, a test is written for a behavior that does not exist yet. The test fails; it may not even compile, since the method has not been written yet. In the **green** step, the simplest code that will make the test pass is written. The code does not have to be pretty at this step; it is enough for the test to pass. In the **refactoring** step, the code is cleaned up: duplication is removed and names are fixed. As long as the tests stay green, the behavior is preserved.

Example 7.5

How is the `letterGrade` method developed with TDD?

Cycle 1, red: the test `assertEquals("CC", Grade.letterGrade(50))` is written. The `Grade` class does not exist yet; the test does not compile.

Cycle 1, green: the simplest code is written: `return "CC";`. The test passes. The code is obviously incomplete, but the only test we have asks for no more than this.

Cycle 2, red: `assertEquals("FF", Grade.letterGrade(49))` is added. Since the method always returns CC, the new test fails.

Cycle 2, green: `if (score >= 50) return "CC"; return "FF";`. Both tests pass.

Each new grade range is a new cycle. In the end, every line of `letterGrade` has been born from a test, and regression tests accumulate on their own.

The biggest gain of TDD is that it leaves no untested code. A side gain is that it improves design. Because the test is written first, the programmer first thinks of the class as a user would: what should the method be called, which parameters should it take, what should it do on invalid input? A class that is hard to test is usually a badly designed class; TDD makes this visible before any code is written.

7.6 Debugging

A test reports a failure: `expected: <-2> but was: <0>`. But it does not always say which line the defect is in. In a small method, the defect can be found by reading the code; in a program of hundreds of lines, one has to look inside the program while it runs. The tool for this job is the debugger.

7.6.1 The debugger

Definition 7.10

A **debugger** is a tool that lets you inspect the state of a running program. It stops the program at a chosen line, advances it line by line, and shows the values of variables at any moment. The line at which the program is to stop is called a **breakpoint**.

With a debugger, you can do the following:

- run the program command by command (line by line);
- see the source code of the running program together with the current line;
- display the values of variables and, if needed, change them;
- add breakpoints to the program and run it up to a breakpoint;
- examine the state of a crashed or halted program through a dump file.

The last item needs a little explanation. When a program crashes or freezes, its state at that moment can be written to a file. In C, this file is called a *core file*. In Java there are two kinds of dump. A heap dump (an `.hprof` file) contains all the objects on the heap. A thread dump shows which line each thread is at at that moment. A dump is a snapshot of the program; it can be examined without running the program again.



Figure 7.13: A program stopped at a breakpoint. The red dot marks the breakpoint, and the highlighted line shows where the program is stopped at the moment. The panel on the right shows the current values of the variables: `i = 2`, `total = 1`.

Figure 7.13 shows the debugging view of an IDE in simplified form. The program has stopped at the line `total += i;`. The line has not run yet: in the first iteration of the loop `total` became 1, and now the second iteration has begun. A program stopped at a breakpoint waits as long as needed. The

programmer looks at the variables, moves to the next step, or lets the program continue.

IntelliJ IDEA, Eclipse, NetBeans, and VS Code all have a built-in debugger. A breakpoint is set by clicking next to the line number, and the program is started with the *Debug* button. A program started with the ordinary *Run* button does not stop at breakpoints.

A common alternative to the debugger is debugging with `println`: lines such as `System.out.println("i = " + i)` are added at suspicious places. This method is fast and works everywhere. But it clutters the code, requires recompiling after every change, and the added lines are often forgotten and never removed. The debugger, on the other hand, does not change the code at all; which variable to watch is decided while the program is running.

7.6.2 Breakpoints and stepping commands

Once the program has stopped at a breakpoint, it is advanced with stepping commands. The command names are in English in IDEs; the shortcuts vary from one IDE to another.

Command	What does it do?	IntelliJ	Eclipse
Breakpoint	The program stops when it reaches this line, *before the line runs*	Ctrl+F8	Ctrl+Shift+B
Step Over	Runs the line; if it contains a method call, completes the call without entering it	F8	F6
Step Into	Enters the method call on the line	F7	F5
Step Out	Finishes the current method and returns to the caller	Shift+F8	F7
Resume	Runs until the next breakpoint	F9	F8
Evaluate / Watch	Evaluates an expression (e.g. `arr[i] > best`) on the spot, or watches it continuously	Alt+F8	Ctrl+Shift+I

Table 7.12: Debugger commands and their default shortcuts.

The commands in Table 7.12 that are most often confused are Step Over and Step Into. Both run the current line. The difference appears when the line contains a method call. Step Into enters that method and stops at its first line. Step Over runs the method in a single step and stops at the next line. If you are sure the called method works correctly, Step Over saves time. If you suspect the defect is inside that method, use Step Into.

The shortcuts follow the default key mapping; the same commands are also reachable from the menu (Run / Debug). There is no need to memorize the shortcuts in the table. It is enough to know what the commands do.

The ordinary breakpoint has two powerful variants. A **conditional breakpoint** stops only if the given condition is true. If a loop runs 10,000 times and the defect shows up only when `i` is 999, the condition `i == 999` is written on the breakpoint. The program runs through 999 iterations without

stopping and stops at exactly the moment you are after. An **exception breakpoint** stops at the moment a particular exception (e.g. `NullPointerException`) is thrown. Because it stops where the exception is thrown rather than where it is caught, the current variables have not yet been lost.

7.6.3 Code trace: finding the defect with the debugger

This time let us approach the failure in the fault-chain example with the debugger. Without reading the code, starting only from the failure `Highest: 0` on the screen, we want to find the defect.

```
public class FindMax {
  public static int max(int[] arr) {
    int best = 0;
    for (int i = 0; i < arr.length; i++) {
      if (arr[i] > best) {
        best = arr[i];
      }
    }
    return best;
  }

  public static void main(String[] args) {
    int[] temps = {-5, -2, -8};
    int result = max(temps);
    System.out.println("Max: " + result);
  }
}
```

A breakpoint is set on line 5, on the condition `if (arr[i] > best)`. Another breakpoint is set on the first line of `main` (13), and the program is started with Debug. The program stops at line 13; one Step Over brings us to the call `max(temps)` on line 14. Step Into enters `max`. A new frame is added to the stack; the `arr` parameter in this frame points to the very same array as `temps` in `main`. The array has not been copied; only its reference has been passed. Resume advances the program to the breakpoint.

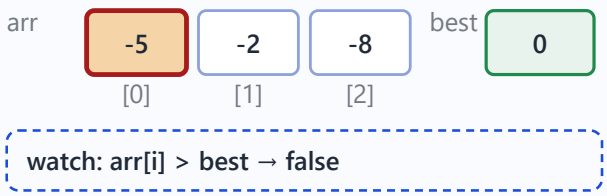


Figure 7.14: The first stop. `i = 0`, `arr[0] = -5`, `best = 0`. The watched expression `arr[i] > best` evaluates to `false`.

The expression `arr[i] > best` is added to the watch list. The debugger recomputes this expression at every stop. At the first stop the expression is false: −5 is not greater than 0 (Figure 7.14). Resume moves on to the later iterations of the loop. For `i = 1`, −2 > 0 is again false; `best` does not change. For `i = 2`, −8 > 0 is false too. At this point the suspicion becomes clear: `best` is never updated.



Figure 7.15: The loop has ended. `best` is still 0, and 0 is returned. The defect is found: the initial value 0 is wrong if all elements are negative.

Step	Command	Line	`i`	`arr[i]`	`best`	`arr[i] > best`
1	Debug	13 (`main`)	—	—	—	—
2	Step Over	14 (`main`)	—	—	—	—
3	Step Into	3	—	—	not yet	—
4	Resume	5	0	−5	0	`false`
5	Resume	5	1	−2	0	`false`
6	Resume	5	2	−8	0	`false`
7	Step Over (several times)	9 (`return`)	—	—	0	—

Table 7.13: The steps of the debugging session.

Table 7.13 summarizes the session. In the third step a new frame is opened; `best` has not been assigned yet. In the last step the loop ends and we reach the line `return best;`; the value to be returned is 0. The location of the defect is now clear: `int best = 0;`. The correct line is `int best = arr[0];`. When the program is allowed to continue, the failure appears on the screen: `Highest: 0`.

After the fix, this case is added as a unit test. Every defect found with the debugger is made permanent by a test that catches it. This way, if the same defect comes back, there is no need for the debugger again; the test reports it at once.

7.6.4 Step Into, Step Over, Step Out

Let us trace the difference among the three stepping commands in a program with nested method calls.

```
public class StepDemo {
  static int square(int x) {
    return x * x;
  }

  static int sumSquares(int a, int b) {
    int sq1 = square(a);
    int sq2 = square(b);
    return sq1 + sq2;
  }

  public static void main(String[] args) {
    int s = sumSquares(3, 4);
    System.out.println("s = " + s);
  }
}
```

The line numbers are as follows: line 1 is the class header, line 3 is `return x * x;`, line 7 is `int sq1 = square(a);`, line 8 is `int sq2 = square(b);`, line 9 is `return sq1 + sq2;`, line 13 is `int s = sumSquares(3, 4);`, line 14 is `println`, and line 15 is the closing brace of `main`. The breakpoint is on line 13, and the program has stopped there. The line has not run yet.

Command	Line reached	Frames on the stack	Visible variables	Explanation
(breakpoint)	13	`main`	—	the line has not run yet
Step Into	7	`main` → `sumSquares`	`a = 3`, `b = 4`	entered `sumSquares`
Step Over	8	`main` → `sumSquares`	`a`, `b`, `sq1 = 9`	`square(3)` ran but was not entered
Step Into	3	`main` → `sumSquares` → `square`	`x = 4`	this time we are inside `square(4)`
Step Out	8	`main` → `sumSquares`	`a`, `b`, `sq1 = 9`	`square` returned 16; the assignment is about to complete
Step Over	9	`main` → `sumSquares`	`sq1 = 9`, `sq2 = 16`	the assignment completed
Step Out	13	`main`	—	`sumSquares` returned 25
Step Over	14	`main`	`s = 25`	the assignment completed
Step Over	15	`main`	`s = 25`	the output was printed: `s = 25`

Table 7.14: The effect of the stepping commands in the `StepDemo` program.

Table 7.14 shows the difference among the three commands clearly. The first Step Into enters the call to `sumSquares` and adds a new frame to the stack. The chain of frames in the third column is the call stack. The Step Over that follows does not enter the call `square(3)` on line 7; it runs the call as a whole and moves to line 8. `sq1` is now 9.

On line 8, this time Step Into is chosen, and we enter `square(4)`. There are three frames on the stack. In the `square` frame only `x` is visible; `a`, `b`, and `sq1` are in another frame. Step Out runs the rest of `square` in a single step and returns to the calling line. The subtle point to notice is this: after Step Out we are still on line 8. The method has returned 16, but this value has not yet been assigned to `sq2`. The assignment completes only at the next Step Over.

The second Step Out finishes `sumSquares` and returns to line 13 in `main`; the return value is 25. The last two Step Overs complete first the assignment (`s = 25`) and then the output. After that, Resume ends the program.

Example 7.6

The program is stopped on line 7 (`int sq1 = square(a);`) and `a = 3`. If two Step Overs are made in a row, where are we and what are the variables?

The first Step Over runs `square(3)` without entering it and completes the assignment: line 8, `sq1 = 9`.

The second Step Over runs `square(4)`: line 9, `sq1 = 9`, `sq2 = 16`.

We never enter `square`. There are always two frames on the stack: `main` and `sumSquares`.

7.6.5 Debugging from the command line: `jdb`

The debuggers of IDEs use the same mechanism behind the scenes: the JVM's debugging interface. The JDK ships with a tool that uses this interface from the command line: `jdb` (the Java Debugger). `jdb` does with commands what IDEs do with buttons. It is useful on a server with no IDE installed or on a machine with only terminal access.

```
public class Jdb {
  static int square(int x) {
    int result = x * x;
    return result;
  }

  public static void main(String[] args) {
    int total = 0;
    for (int i = 1; i <= 3; i++) {
      total += square(i);
    }
    System.out.println("Sum: " + total);
  }
}
```

The program adds up the squares of the numbers from 1 to 3: 1 + 4 + 9 = 14. To run it under the debugger, it is first compiled with the `-g` option and then started with `jdb` instead of `java`:

```
javac -g Jdb.java
jdb Jdb
```

The `-g` option also writes the names of local variables into the class file. In a class compiled without this option, the `locals` command cannot show variable names, because the class file contains only slot numbers.

Command	Meaning
`stop at Class:line`	sets a breakpoint
`clear Class:line`	removes a breakpoint
`run` / `cont`	starts / continues
`step` / `next` / `step up`	Step Into / Over / Out
`where`	shows the call stack
`locals`, `print x`	shows the local variables, or a single value

Table 7.15: Basic `jdb` commands.

The commands in Table 7.15 map one to one onto the IDE commands. `step` corresponds to Step Into, `next` to Step Over, `step up` to Step Out, and `cont` to Resume.

`jdb` can also attach to a running application. For this, the application is started with a special option: `java -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005 MyApp`. This option makes the JVM load the debugging agent and listen on network port 5005. `suspend=y` makes the program wait until a debugger attaches. Then the command `jdb -attach 5005`, or an IDE, connects to this port. This is how one debugs a program on a remote server.

7.6.6 A `jdb` session

The session below was captured by running it with JDK 21; the lines that start with `>` and `main[1]` are the `jdb` prompt.

```
> stop at Jdb:4
Deferring breakpoint Jdb:4.
It will be set after the class is loaded.
> run
VM Started: Set deferred breakpoint Jdb:4
Breakpoint hit: "thread=main", Jdb.square(), line=4 bci=4
main[1] where
  [1] Jdb.square (Jdb.java:4)
  [2] Jdb.main (Jdb.java:10)
main[1] locals
Method arguments:
x = 1
Local variables:
result = 1
```

The command `stop at Jdb:4` sets a breakpoint on line 4, on `return result;`. Because the class has not been loaded yet, `jdb` defers the breakpoint and places it when the class is loaded. `run` starts the program. The program stops when it reaches line 4. The `bci=4` in the message shows at which byte of the method's bytecode we are stopped; when working at the source level, this number does not matter.

The `where` command shows the call stack: `square` (line 4) on top, and below it the `main` (line 10) that called it. `locals` lists the variables in the current frame. `x = 1` is the method argument, and `result = 1` is the local variable. This is the first iteration: `square(1)` has been called for `i = 1`.

```
main[1] cont
Breakpoint hit: "thread=main", Jdb.square(), line=4 bci=4
main[1] print result
 result = 4
main[1] clear Jdb:4
Removed: breakpoint Jdb:4
main[1] cont
> Sum: 14
```

`cont` lets the program continue. In the second iteration of the loop `square(2)` is called, and the program stops again at the same breakpoint. This time `print result` shows 4. `clear Jdb:4` removes the breakpoint; with the last `cont`, the program runs to the end without stopping again and prints `Sum: 14`.

Command	Stopped at	`i` (`main`)	`x`	`result`	`total`
`run`	`square`, line 4	1	1	1	0
`cont`	`square`, line 4	2	2	4	1
`clear` + `cont`	the program ends	—	—	—	14

Table 7.16: The state of the program in the `jdb` session.

The `total` column in Table 7.16 is the value in `main` at the moment of stopping at the breakpoint. At the second stop, `total` is 1, because the result of the first iteration has been added and that of the second has not yet been. Had the breakpoint been set on line 3 (`int result = x * x;`), the `locals` command would not have shown `result` : since the line has not run yet, the variable does not exist at that moment.

7.6.7 Systematic debugging

The debugger is a powerful tool, but it is not a method on its own. Setting random breakpoints and changing the code at random can take hours. Experienced programmers follow a five-step path.

1. **Reproduce.** Find the smallest input that triggers the failure every time, for example `{-5, -2, -8}` . A bug that cannot be reproduced cannot be debugged.
2. **Write it as a test.** Write a unit test that fails on this input. The red test is the proof of the bug, and it will also show that the fix worked.
3. **Narrow it down.** Start from the first line in the stack trace. Watch the variables at breakpoints and shrink the suspicious region by halving it. The place you are looking for is the line where a value first becomes wrong.
4. **Form a hypothesis and test it.** Propose an explanation such as " `best` is never updated because it starts at 0" and confirm it in the debugger. Changing code by guesswork creates new bugs.
5. **Fix and run all the tests.** The new test must turn green and the old ones must stay green; that is, there must be no regression. The test stays in the test suite permanently.

The idea of "halving" in the third step resembles binary search. If the result of a 200-line computation is wrong, a breakpoint is first set on the middle line, line 100. If the values are correct there, the defect is in the second half; if they are wrong, it is in the first half. At each step the region being searched is halved. 200 lines shrink to a single line in at most eight steps, since $2^8 = 256 \geq 200$.

Another tool that helps with narrowing is Java's `assert` statement. `assert` lets the programmer write into the code the assumption "at this point this condition is always true." If the condition is false, the program throws `AssertionError` , and the problem is caught where it occurs.

```
static double average(int[] d) {
  assert d.length > 0 : "empty array";
  int t = 0;
  for (int v : d) {
    t += v;
  }
  return (double) t / d.length;
}
```

```
public static void main(String[] a) {
  System.out.println(average(new int[] {1, 2}));
  System.out.println(average(new int[0]));
}
```

```
$ java As
1.5
NaN
$ java -ea As
1.5
Exception in thread "main" java.lang.AssertionError: empty array
        at As.average(As.java:3)
        at As.main(As.java:12)
```

The output shows an important detail: `assert` statements are disabled by default. The program run with plain `java As` gives no warning for the empty array. The division 0 / 0 produces `NaN` (not a number) in `double`; the program silently prints a wrong value. When it is run with the `-ea` (enable assertions) option, the assumption is checked. The moment the program is called with an empty array, it stops at line 3 with `AssertionError: empty array`. For this reason, `assert` is turned on during development and testing and left off in the program that goes to the user. To check user input, use an exception, not `assert`; a disabled `assert` prevents nothing.

7.7 Analysis tools

Unit tests and the debugger are the two tools programmers use most often, but they do not answer every question. Questions such as "Why is the program slow?", "How much of the code do my tests exercise?", and "Why is memory filling up?" need other tools. Each of the tools in this section focuses on a single question.

7.7.1 The profiler

Definition 7.11

A **profiler** is a tool that measures where a program spends its time and memory. It shows which method was called how many times, how long it took, and how much memory it allocated.

A profiler presents the call graph, which shows who calls whom, in several forms: a flat list, a tree, or a flame graph. In a flame graph, each method is a box; the width of the box is proportional to the time spent in that method. Wide boxes stand out immediately. The technical details of the measurement

vary with the compiler and the JVM version in use.

The basic rule of profiling is short: measure first, then optimize. Most of a program's time is usually spent in a small region, a *hot spot*. The programmer's guess often misses this spot. An optimization made by guesswork complicates the code and speeds up the wrong place.

Old JVMs had a built-in profiler that was switched on with the `java -Xprof` option. This option was removed in Java 10. In Java 21, the same option only produces a warning and is ignored:

```
OpenJDK 64-Bit Server VM warning: Ignoring option -Xprof; support was removed in 10.0
```

Today other tools are used in Java. The **JDK Flight Recorder** (JFR) that ships with the JDK records events, and **JDK Mission Control** visualizes these recordings. To these are added the free **VisualVM** and the built-in profilers of IDEs. JFR records events while the program runs, slowing it down very little. The program is run with a JFR recording like this, and a summary of the recording is obtained like this:

```
java -XX:StartFlightRecording=filename=rec.jfr Profile
jfr summary rec.jfr
```

The first command runs the program and writes the events to the file `rec.jfr`. The second command lists the event types and counts in the recording. A detailed analysis is usually done by opening this file in JDK Mission Control.

7.7.2 Measuring a hot spot

To see the value of profiling, two methods that do the same job in two different ways are compared. Both build a string by writing the numbers from 0 to n − 1 one after another.

```
public class Profile {
  static String concatPlus(int n) {
    String s = "";
    for (int i = 0; i < n; i++) {
      s += i;  // new String each pass
    }
    return s;
  }

  static String concatBuilder(int n) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < n; i++) {
      sb.append(i);
    }
    return sb.toString();
  }
  // main: see the next box
}
```

```
public static void main(String[] args) {
  long t0 = System.nanoTime();
  String a = concatPlus(50_000);
  long t1 = System.nanoTime();
  String b = concatBuilder(50_000);
  long t2 = System.nanoTime();
  long ms1 = (t1 - t0) / 1_000_000;
  long ms2 = (t2 - t1) / 1_000_000;
  System.out.println("+= : " + ms1 + " ms");
  System.out.println("SB : " + ms2 + " ms");
  System.out.println(a.equals(b));
}
```

The `main` method measures the time of the two methods with `System.nanoTime()`. This method returns a timestamp in nanoseconds; the difference of two timestamps is divided by 1,000,000 to convert it to milliseconds. The last line verifies that the two strings are identical: the speed difference must not change the result. In one run the output is as follows; the numbers vary with the machine and the run:

```
+= : 873 ms
SB : 2 ms
true
```

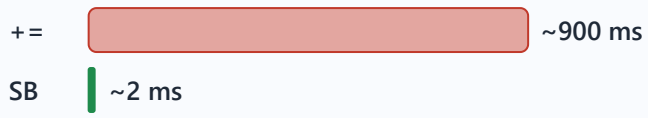


Figure 7.16: Running times: about 900 milliseconds with `+=`, about 2 milliseconds with `StringBuilder`.

The difference is hundreds of times (Figure 7.16). While this note was being prepared, in three runs with JDK 21 the `+=` version took 862–922 ms and the `StringBuilder` version 2–4 ms. The reason is that `String` is immutable. `s += i` cannot modify the old string; it creates a new `String` in every iteration. All the characters of the old string are copied into this new object. In iteration k, the number of characters copied is roughly proportional to k. The total work grows as $1 + 2 + \ldots + n \approx n^2/2$: $O(n^2)$. `StringBuilder`, on the other hand, appends to a single buffer. When the buffer fills up, it is moved to a larger one, but since the size doubles each time, the total copying stays O(n).

In this program the hot spot is obvious. In a real program, however, it is not visible at first glance: the line `s += i` sits in an innocent-looking loop among hundreds of lines. A profiler shows that 99% of the time is spent on this line and tells you where to make the improvement.

7.7.3 Code coverage

> **Definition 7.12**
>
> **Code coverage** measures which parts of the code are executed while the tests run. The result is usually given as a ratio: the ratio of executed lines or branches to the total.

A code coverage tool reveals pieces of code that are never executed or used, and helps find untested regions. For this reason, it is an important tool for increasing the reliability of software. In Java, the most common tool is **JaCoCo**, which works with Maven and Gradle. The "Run with Coverage" command in IntelliJ and the **EclEmma** plugin in Eclipse do the same job. The result is shown by coloring the lines of code: executed lines green, never executed ones red.

Criterion	What is counted?
Line	Lines executed at least once
Branch	Both the true and the false outcome of every `if` / loop condition
Method	Methods called at least once

Table 7.17: Code coverage criteria.

Of the three criteria in Table 7.17, the strictest is branch coverage. When an `if` line runs, line coverage counts it as "executed." Branch coverage, in contrast, wants to see the condition come out both true and false. If the condition has always been true, a defect hiding in the false branch has never been tried.

One must know the limit of coverage well: 100% coverage does not mean defect-free code. Coverage says only that a line was executed, not that its result was correct. A test that contains no assertion and merely calls the method also raises coverage. A coverage report is very valuable for the warning "this was never tried"; as for "this is correct," it says nothing.

7.7.4 Code trace: code coverage

The example method returns the sign of a number as a string.

```
public class Sign {
  public static String sign(int x) {
    if (x > 0) {
      return "positive";
    } else if (x < 0) {
      return "negative";
    }
    return "zero";
  }
}
```

The executable lines of the method are 3, 4, 5, 6, and 8. Lines 1, 2, and 7 are declarations or closing braces; coverage tools do not count them. There are two conditions: `x > 0` and `x < 0`. Since each condition has two branches, 4 branches are counted in total. Two tests have been written for the method:

```
class SignTest {
  @Test
  void positive() {
    assertEquals("positive", Sign.sign(5));
  }

  @Test
  void negative() {
    assertEquals("negative", Sign.sign(-3));
  }
}
```

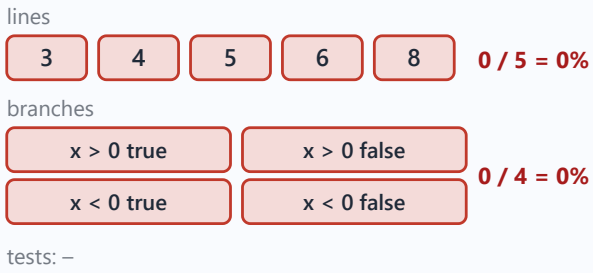


Figure 7.17: No test has run. All five executable lines and all four branches are red.

In the call `sign(5)`, line 3 runs and `x > 0` is true; line 4 returns `"positive"`. The test passes. Coverage: 2 of 5 lines, 1 of 4 branches. In the call `sign(-3)`, the condition on line 3 is false (a new branch), `x < 0` on line 5 is true (a new branch), and line 6 returns `"negative"`. Both tests are green.

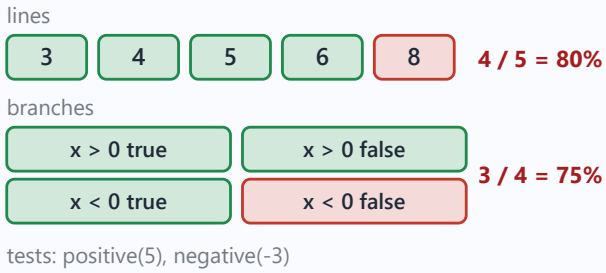


Figure 7.18: The report after two tests: line coverage 4 / 5 = 80%, branch coverage 3 / 4 = 75%. Line 8 and the false branch of `x < 0` have never been tried.

Test	Lines executed	Branches tried	Line coverage	Branch coverage
(none)	—	—	0 / 5	0 / 4
`sign(5)`	3, 4	`x > 0` true	2 / 5 = 40%	1 / 4 = 25%
+ `sign(-3)`	3, 4, 5, 6	+ `x > 0` false, `x < 0` true	4 / 5 = 80%	3 / 4 = 75%
+ `sign(0)`	3, 4, 5, 6, 8	+ `x < 0` false	5 / 5 = 100%	4 / 4 = 100%

Table 7.18: Line and branch coverage after each test.

Table 7.18 shows the missing test clearly. Despite two green tests, line 8 and the false branch of the condition `x < 0` have never been tried. Only an input for which both conditions are false, that is 0, reaches this branch. The coverage report thus exposes the forgotten boundary value: `sign(0)`. When this test is added, line and branch coverage become 100%, and the test verifies that the method returns `"zero"`.

Even at 100% coverage, it is not coverage but `assertEquals` that checks correctness. Had line 8 mistakenly been `return "positive";`, a test that calls `sign(0)` without looking at the result would still bring coverage to 100%.

7.7.5 Static analysis: FindBugs and SpotBugs

> **Definition 7.13**
>
> **Static analysis** is an inspection that looks for bug patterns without running the code. The word "static" says that the program is not running; tests and the debugger, in contrast, run the program and are *dynamic* tools.

FindBugs was a tool developed for the static analysis of Java code. Without running the program, it inspected the compiled code and looked for known bug patterns: `null` dereferences, wrong comparisons, resources that are never closed. FindBugs is no longer developed; its successor is **SpotBugs**. SpotBugs is used through Maven, Gradle, and IDE plugins. Similar tools include PMD, Checkstyle, and SonarQube.

```
import java.util.Scanner;

public class SpotDemo {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    String answer = sc.nextLine();
    // result ignored:
    answer.trim();
    // String compared with ==:
    if (answer == "yes") {
      System.out.println("Approved");
    } else {
      System.out.println("Rejected");
    }
  }
}
```

The program reads an answer from the user and approves if it is `"yes"`. Even though the user types `yes`, the output is `Rejected`. The program has two defects, and the compiler stays silent about both.

SpotBugs pattern	Warning
`ES_COMPARING_STRINGS_WITH_EQ`	`String` objects are compared with `==`; the correct way is `equals`
`RV_RETURN_VALUE_IGNORED`	`trim()` returns a new `String`; the result is discarded

Table 7.19: The patterns SpotBugs flags in this program.

The first pattern in Table 7.19 is a reminder that `==` checks whether two references point to the same object. `nextLine()` creates a new `String` object for the text read from the keyboard. This object is not the same object as the `"yes"` constant in the code; even if their contents are the same, `==` returns false. The correct form is `"yes".equals(answer)`. The second pattern concerns the

immutability of `String`. `answer.trim()` does not change `answer`; it returns a new string with the leading and trailing whitespace removed. Because the result is not assigned to a variable, the line does nothing. The correct form is `answer = answer.trim();`.

A test could also have found these two defects, but only if it had been written with the right input. Static analysis, on the other hand, needs no input: it warns wherever it sees the pattern. This is why it complements tests rather than replacing them. Static analysis cannot know about logic errors; a defect such as "the difference is computed instead of the sum" matches no pattern.

7.7.6 The first static analyzer: the compiler

Every Java programmer already has a static analyzer at hand: `javac`. The compiler inspects the code without running it and reports compile errors. With the `-Xlint` option, it also reports constructs that are not compile errors but look suspicious.

```
public class Lint {
  public static void main(String[] args) {
    int day = 2;
    switch (day) {
      case 1:
        System.out.println("Monday");
      case 2:
        System.out.println("Tuesday");
      case 3:
        System.out.println("Wednesday");
        break;
    }
    int x = 10 / 0;
  }
}
```

The program compiles and runs, but it contains two defects. The `break` has been forgotten at the end of `case 1` and `case 2`. On line 13 there is also a division by a constant zero. When run, the output is as follows:

```
Tuesday
Wednesday
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Lint.main(Lint.java:13)
```

For `day = 2`, execution enters at `case 2`. Since there is no `break`, control falls through to `case 3`, and `Wednesday` is printed as well. Then the division on line 13 throws `ArithmeticException`. Plain `javac Lint.java` compiles this program without any warning. When it is compiled with `-Xlint:all`, both defects become visible without running the program:

```
Lint.java:13: warning: [divzero] division by zero
    int x = 10 / 0;
                ^
Lint.java:7: warning: [fallthrough] possible fall-through into case
      case 2:
      ^
Lint.java:9: warning: [fallthrough] possible fall-through into case
      case 3:
      ^
3 warnings
```

The `[divzero]` warning reports the division by a constant zero, and the two `[fallthrough]` warnings report that the previous `case` does not end with a `break`. The order of the warnings follows the compiler's internal phases, not the line order. There is no warning for `case 1`, because no `case` precedes it. Reading warnings is the cheapest way of finding a bug before running the program.

7.7.7 Property-based testing

In unit tests, the programmer chooses the inputs. However carefully they are chosen, there may be a case that did not occur to the programmer. Property-based testing solves this problem by letting the computer choose the inputs.

Definition 7.14

Property-based testing is testing that checks, with a large number of randomly generated inputs, that a property always holds. Instead of individual examples, the programmer writes a **property**: "this must be true for every input." If an input that breaks the property is found, it is reported as a **counterexample**.

The pioneer of the method is **QuickCheck**. QuickCheck generates the tests automatically: it is the tool, not the programmer, that picks the inputs at random. When a counterexample is found, the tool shrinks it where possible; this way a counterexample with 2 elements is reported instead of an array of 50 elements. The first QuickCheck was written for the Haskell language. Versions exist for many languages; in Java they are **jqwik** and **junit-quickcheck**. The test below applies the same idea with plain JUnit.

```
class PropertyTest {

  static String randomText(Random r) {
    int len = r.nextInt(20);      // 0..19
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < len; i++) {
      sb.append((char) ('a' + r.nextInt(26)));
    }
    return sb.toString();
  }

  @Test
  void reverseTwiceGivesSame() {
    Random r = new Random(42);    // fixed seed
    for (int k = 0; k < 1000; k++) {
      String s = randomText(r);
      String t = TextUtils.reverse(s);
      assertEquals(s, TextUtils.reverse(t));
    }
  }
}
```

The method under test is a simple helper that reverses a string:

```
public class TextUtils {
  public static String reverse(String s) {
    return new StringBuilder(s).reverse().toString();
  }
}
```

The property being tested is this: reversing a string twice gives the string itself. `randomText` produces a random string of length 0–19 made of lowercase letters. `r.nextInt(26)` gives a number between 0 and 25; added to `'a'` and converted to `char`, it becomes a letter from `a` to `z`. The test generates 1000 such strings and verifies the property for each. The fixed seed (42) makes the test deterministic: the same 1000 strings are tried in every run. When the test is run, the report is a single line: 1 test, 1000 inputs, passed.

This test tries something different from hand-picked examples. Cases such as the empty string (length 0), a one-letter string, or a repeated letter arise on their own in random generation. The programmer does not have to think of them one by one.

7.7.8 The property test finds the bug in the first version

The property test can also be applied to the first, faulty version of the `max` method. The largest element has two properties: the result is not smaller than any element of the array, and the result is an element of the array.

```
class PropertyTest2 {

  @Test
  void maxIsLargestElement() {
    Random r = new Random(7);
    for (int k = 0; k < 1000; k++) {
      int[] arr = new int[1 + r.nextInt(5)];
      for (int i = 0; i < arr.length; i++) {
        arr[i] = r.nextInt(201) - 100;      // -100..100
      }
      int best = ArrayUtils.max(arr);
      boolean found = false;
      for (int x : arr) {
        assertTrue(best >= x);              // property 1
        if (x == best) found = true;
      }
      assertTrue(found,                       // property 2
        "Counterexample: " + Arrays.toString(arr));
    }
  }
}
```

The test generates 1000 random arrays with seed 7. The length of each array is 1–5, and its elements are between −100 and 100: `r.nextInt(201)` gives 0–200, and subtracting 100 makes the range −100..100. For each array, property 1 (`best >= x`) is checked first, then property 2 (`found`). Run with the first, faulty version, the result is this:

```
✗ maxIsLargestElement()
  Counterexample: [-60, -69, -39, -65, -13] ==> expected: <true> but was: <false>
1 test · 1 failed
```

The counterexample is the second random array; the first array contains a non-negative element, and the faulty version gives the right result. All the elements of the second array are negative; the faulty version returns 0. Property 1 holds even in the faulty version: 0 is greater than all the negative numbers. Property 2, however, catches the bug: 0 is not in the array. The message, which is the second parameter of `assertTrue`, writes the counterexample into the report.

When the same test is run with the corrected version, all 1000 inputs pass. The lesson here is important: properties must be written completely. Had only property 1 been written, the test would have approved the faulty code too. A weak property gives a false sense of confidence.

Example 7.7

Suppose the property "in the result array, no element is greater than the next one" is written for a sorting method. Is this property enough on its own?

It is not enough. A method that ignores its input and always returns an empty array also satisfies this property. So does a method that returns an array of all zeros. Both produce a sorted array, but it has nothing to do with the input.

The missing property is this: the result must be a permutation of the input; the same elements, the same number of times. Together, the two properties define the notion of "sorting" completely.

7.7.9 Selenium

Selenium is a framework used to test web applications. It makes browser-based tests easier: it opens a real browser from a program, clicks a link, types into a form, presses a button, and verifies that the expected text appears on the page. It is used with languages such as Java, Python, C#, and JavaScript. In Java it is usually called from inside a JUnit test.

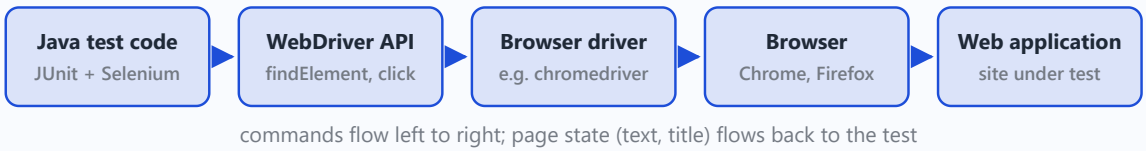


Figure 7.19: The Selenium architecture. The Java test code calls the WebDriver interface; the commands reach the real browser through the browser driver, and from there the web application.

Figure 7.19 shows the path the commands follow. The test code uses a common interface called WebDriver. Each browser has its own driver (ChromeDriver for Chrome, GeckoDriver for Firefox). The same test code can be run in different browsers by swapping the driver. A typical scenario goes like this: the login page is opened, a username and password are typed, the "Log in" button is pressed, and one checks whether the text "Welcome" appears in the title.

Selenium tests check the whole that the user sees; they are at the level of system and acceptance tests. This power has a price. They are much slower than unit tests: even opening a browser takes seconds. They are also brittle: even a change in the name of a button can break the test. For this reason, their number is kept small; they stand at the top of the test pyramid.

7.7.10 Memory leaks

In Java, memory is managed automatically thanks to the garbage collector. The programmer creates objects with `new` but never explicitly deletes them. Even so, Java programs can leak memory too.

Definition 7.15

The **garbage collector** (GC) is the JVM component that reclaims the memory of objects that no reference points to. A **memory leak**, on the other hand, is memory that is not freed because a reference to objects that are no longer needed still remains.

The key concept behind this definition is reachability. The garbage collector removes only unreachable objects. If a chain of references from anywhere in the program, for example from a `static` field, reaches an object, that object is not removed even if it is unnecessary for the program.

```
import java.util.ArrayList;
import java.util.List;

public class Leak {
  static List<byte[]> cache = new ArrayList<>();

  static void handle(int id) {
    byte[] data = new byte[1024 * 1024];  // 1 MB
    cache.add(data);                      // never removed!
  }

  public static void main(String[] args) {
    for (int i = 1; ; i++) {
      handle(i);
      if (i % 20 == 0) {
        System.out.println(i + " requests handled");
      }
    }
  }
}
```

The program handles requests in an infinite loop. Each request creates a 1 MB array and adds it to a `static` list. The list is never cleared.

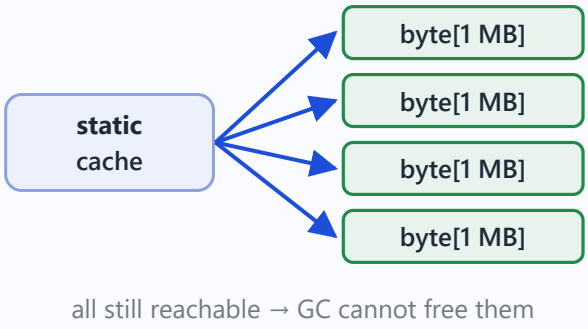


Figure 7.20: The `static` `cache` list holds the 1 MB arrays added on each request. All of the arrays are still reachable; the garbage collector cannot free them.

When the program is run with the command `java -Xmx256m Leak`, the output is as follows; the lines in between have been abbreviated:

```
20 requests handled
…
120 requests handled
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at Leak.handle(Leak.java:8)
    at Leak.main(Leak.java:14)
```

The `-Xmx256m` option limits the heap to at most 256 MB. After handling 120 requests, the program crashes after a while with `OutOfMemoryError: Java heap space`. The stack trace points to the right place of the problem: `new byte[1024 * 1024]` on line 8. But the real defect is not on this line. Memory runs out on that line because none of the previous arrays has been released. The defect is `cache.add(data)` on line 9: the data that is added is never removed again.

It may seem surprising that only about 120 arrays of 1 MB fit in a 256 MB heap. The reason is that G1, the JVM's default garbage collector, divides memory into regions. At this heap size, regions are 1 MB. A 1 MB array, together with its object header, does not fit in one region and occupies two. This is why the heap runs out when it is about half full. The exact number varies with the JVM and the settings.

7.7.11 Eclipse Memory Analyzer and VisualVM

The way to find a leak is to look at the heap: which objects exist, how much space do they take, and who is holding them? **Eclipse Memory Analyzer** (MAT) answers these questions. It analyzes the Java heap to find memory leaks and helps with memory optimization.

The input of MAT is a **heap dump** (an `.hprof` file): a snapshot of all the objects at a given moment. A dump is taken in two ways:

```
// on OutOfMemoryError
java -XX:+HeapDumpOnOutOfMemoryError Leak
// from a running program
jps
jcmd <pid> GC.heap_dump heap.hprof
```

The first way is the `-XX:+HeapDumpOnOutOfMemoryError` option. A JVM started with this option dumps the heap to a file by itself at the moment memory runs out. The second way is to take a dump from a running program. `jps` lists the running JVMs and their process IDs (pid). `jcmd <pid> GC.heap_dump heap.hprof` writes the heap of that JVM to a file.

The three most important reports of MAT are these. **Leak Suspects** automatically finds the objects that hold a large part of the heap. The **Dominator Tree** shows which object holds the most memory; all the objects that would be freed together with an object if it were removed are listed under it. The

Histogram gives the number of objects per class and the space they occupy.

In the `Leak` example, the Leak Suspects report shows that almost the whole heap is held by the `Leak.cache` list and the `byte[]` arrays inside it. The fix is one of two ways: data that is finished with is removed from the list, or a size-limited cache is used. A bounded cache discards its oldest element when it fills up, and memory use stays constant.

VisualVM watches a running program live. It visualizes memory consumption, thread usage, CPU load, and more. It connects to running JVMs, takes profiles, and takes heap and thread dumps. VisualVM is free and open source (visualvm.github.io); it used to ship with the JDK under the name `jvisualvm`. Tools that give similar information on the command line are `jps`, `jstat`, `jcmd`, and `jconsole`.

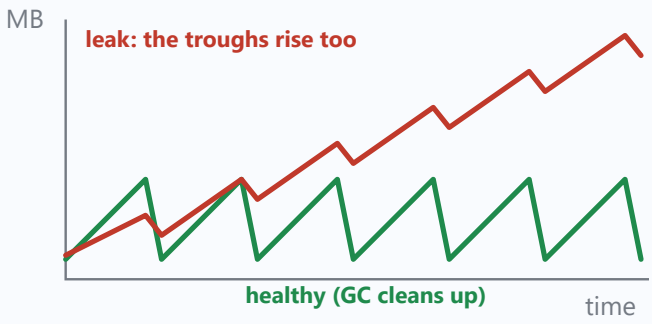


Figure 7.21: Used heap memory over time. In a healthy program it is a sawtooth: garbage collection brings memory down to the same trough every time. With a leak, the troughs keep rising.

Figure 7.21 shows two typical views of the heap graph in VisualVM. In a healthy program, memory use rises, garbage collection kicks in, and the use falls back to the same level; the graph looks like a sawtooth. With a leak there is also a sawtooth, but the trough after each garbage collection is higher than the previous one. If the troughs keep rising, objects are not being released. This picture is the first sign of a leak. Then a heap dump is taken and MAT is used to examine who is holding what.

7.7.12 Which tool for which question?

Each of the tools described in this note answers a single question. Choosing the right tool starts with putting the question correctly. Table 7.20 matches questions with tools.

Question	Technique / tool
Does this method give the correct result?	Unit test: JUnit, `@Test`, `assertEquals`, `assertThrows`
Do the parts work together, as a whole system, and from the user's point of view?	Module, sub-system, system, acceptance (alpha / beta) testing; Selenium for the web
Did a change break something old?	Regression testing: all automated tests run on every change (CI)

Question	Technique / tool
Which inputs should be chosen?	Boundary values, equivalence classes, `@ParameterizedTest`
Is it correct on the inputs that were not chosen?	Property-based testing (QuickCheck, jqwik)
How much of the code do the tests execute?	Code coverage: JaCoCo, EclEmma
Is there a bug in the code that can be seen without running it?	Static analysis: `javac -Xlint`, SpotBugs (FindBugs)
Why does it work incorrectly?	Debugger: breakpoint, Step Into / Over / Out, watch; `jdb`
Where is the time being spent?	Profiler: JFR, VisualVM
Why is memory filling up?	Watching with VisualVM, examining the heap dump with Eclipse Memory Analyzer

Table 7.20: Which technique or tool for which question?

The tools in the table are not rivals but links in a chain. Regression testing reports a failure. The debugger finds the defect. After the fix, a unit test keeps the defect from coming back. The coverage report shows which tests are missing, and static analysis shows patterns that tests may miss. The profiler and the memory tools, in turn, explain why a correctly working program is slow or greedy. Testable code is usually well-designed code. Taking dependencies from outside, like the `Random` in the die example, makes both testing and design easier.

7.8 Solved problems

The problems in this section combine the topics covered in the note. The solution of each problem is given with its intermediate steps; the JUnit results and messages have been verified by running them with JUnit 5.11.

Example 7.8

Which test passes, and which fails?

```
class QuizTest {
  @Test void a() { assertEquals(7, 3 + 4); }
  @Test void b() { assertEquals(0.3, 0.1 + 0.2); }
  @Test void c() { assertTrue("Java".length() == 4); }
  @Test void d() { assertThrows(ArithmeticException.class,
      () -> System.out.println(10 / 2)); }
}
```

(a) 3 + 4 = 7; expected 7. **Passes.**

(b) 0.1 + 0.2 is 0.30000000000000004 in binary representation. No tolerance is given, so the comparison is exact. **Fails:** `expected: <0.3> but was: <0.30000000000000004>`.

(c) `"Java".length()` is 4, and the condition is true. **Passes.**

(d) 10 / 2 = 5; the division is fine and 5 is printed on the screen. No `ArithmeticException` is thrown. **Fails:** `Expected java.lang.ArithmeticException to be thrown, but nothing was thrown.`

Example 7.9

Branch coverage in an absolute-value method. For the method below, what is the minimum number of tests needed for 100% branch coverage?

```
public class Abs {
  static int abs(int x) {
    if (x < 0) {
      return -x;
    }
    return x;
  }
}
```

The method has a single condition (`x < 0`) and therefore two branches. The true branch needs a negative input (e.g. −3 → 3), and the false branch needs a zero or positive input (e.g. 5 → 5). At least **2 tests** are enough. It would be good to add −1 and 0 as boundaries as well; these two values enclose the threshold from both sides. If the condition had mistakenly been written as `x < -1` , −1 would return −1; the tests for −3 and 5 cannot see this, and only the −1 test catches it.

Example 7.10

What is the problem with this test class?

```
class CounterTest {
  static int count = 0;

  @Test void increment() {
    count++;
    assertEquals(1, count);
  }

  @Test void incrementTwice() {
    count += 2;
    assertEquals(2, count);
  }
}
```

The `count` field is `static`; it belongs to the class, and the test objects of the two tests share the same field. The result depends on the test order. In this class JUnit runs `increment` first: `count` becomes 1 and the test passes. Then `incrementTwice` runs: `count` goes from 1 to 3. **Fails:** `expected: <2> but was: <3>`. If the order were reversed, this time `increment` would fail (`expected: <1> but was: <3>`).

The fix is one of two ways. Remove the word `static`; since JUnit creates a new object for every test, every test starts from 0. Or reset the field in a `@BeforeEach` method.

Example 7.11

Two Step Overs. In the `StepDemo` program, with the debugger stopped on the line `int s = sumSquares(3, 4);` (13), if two Step Overs are made in a row, which line do we reach, what is on the screen, and what is `s`?

The first Step Over runs `sumSquares` without entering it and completes the assignment: `s = 25`. The program is on line 14, at `println`; the screen is still empty.

The second Step Over runs `println`. `s = 25` is printed on the screen, and the program reaches the closing line of `main` (15). `s` is 25.

Example 7.12

Which tool? Which technique or tool should be used in each case?

(a) The program crashes with `OutOfMemoryError` after 10 minutes. The heap graph is watched with VisualVM; if the troughs rise, there is a leak. The heap dump (`-XX:+HeapDumpOnOutOfMemoryError`) is examined with Eclipse Memory Analyzer.

(b) Report generation takes 30 seconds; which method is slow? A profiler: JDK Flight Recorder or VisualVM.

(c) Did the newly added discount code break the old invoice calculation? Regression testing: all the automated tests are rerun, preferably on every change with CI.

(d) Does the registration form on the website work in all browsers? Browser-based tests with Selenium, using different browser drivers.

(e) Are there forgotten `break`s in a `switch`? Static analysis: `javac -Xlint:all` (the `[fallthrough]` warning) or SpotBugs.

(f) A loop produces a wrong value in the 10,000th iteration; what are the variables at that moment? A conditional breakpoint in the debugger: `i == 9999`. Since the loop counter starts at 0, the 10,000th iteration has `i = 9999`.

Example 7.13

Boundaries of a primality check. How are the boundary values 0, 1, 2, 3, 4, 97, and 100 tested with `@ParameterizedTest` for `static boolean isPrime(int n)`?

```
static boolean isPrime(int n) {
  if (n < 2) return false;
  for (int d = 2; d * d <= n; d++) {
    if (n % d == 0) return false;
  }
  return true;
}

@ParameterizedTest
@CsvSource({"0, false", "1, false", "2, true", "3, true",
            "4, false", "97, true", "100, false"})
void primeBoundaries(int n, boolean expected) {
  assertEquals(expected, isPrime(n));
}
```

All seven calls pass. 0 and 1 are not prime; the first line of the method separates these two cases. 2 is the smallest prime and never enters the loop, because $2 \times 2 > 2$. 4 is the smallest composite number; the loop finds a divisor at `d = 2`. 0, 1, and 2 are the boundaries where most mistakes are made: if `n < 1` is written instead of `n < 2`, 1 is taken to be prime. The `d * d <= n` in the loop condition is a boundary too: had `<` been written, 4 would be taken to be prime, because for `d = 2`, $4 < 4$ is false and the loop would never run.

Example 7.14

Testing a rectangle class. Write a JUnit 5 test class for a `Rectangle` class (`width`, `height`, `area()`, `perimeter()`); a negative side must throw `IllegalArgumentException`.

```
class RectangleTest {
  @Test
  void areaCorrect() {
    assertEquals(12.0, new Rectangle(3, 4).area(), 1e-9);
  }

  @Test
  void perimeterCorrect() {
    assertEquals(14.0, new Rectangle(3, 4).perimeter(), 1e-9);
  }

  @Test
  void negativeSideRejected() {
    assertThrows(IllegalArgumentException.class,
      () -> new Rectangle(-1, 2));
  }
}
```

A 3 × 4 rectangle has area 12 and perimeter 2 × (3 + 4) = 14. Since the sides are `double`, a tolerance is given in the assertions. A rectangle with a zero side should also be tried as a boundary: its area is 0, and it must not throw an exception.

Example 7.15

TDD for a transfer method. A method `transfer(BankAccount target, double amount)` is to be added to `BankAccount`. Which tests are written before any code?

(1) Successful transfer: both balances change; the sender's balance decreases and the receiver's increases. (2) Insufficient funds: an exception is thrown and *neither balance changes*. (3) Negative amount: rejected. (4) Transfer to oneself: the agreed behavior (rejection or a no-op) is tested.

The second test is especially important. If the transfer first withdraws from the sender and then deposits to the receiver, and the deposit step throws an exception, the money can be lost in thin air. Each test is first red, and then turns green with the code that is written.

Example 7.16

Integer division in an average. A method that computes the average of an array returns 1.0 instead of 1.5 for `{1, 2}`. How is the defect found with the debugger?

First a failing test is written: for `{1, 2}`, expected 1.5. A breakpoint is set on the `return` line. The variable `total` and the division expression are watched: `total` is 3, which is correct. `total / arr.length`, however, gives 1: since two `int`s are divided, this is an integer division

(3 / 2 = 1). The result is converted to `double` only after the division and becomes 1.0. The fix is `(double) total / arr.length` : the conversion happens before the division, the division becomes floating-point, and gives 1.5. Then all the tests are run.

7.9 Common mistakes

Mistake	Consequence	The right way
Writing expected and actual the wrong way around: `assertEquals(calc.add(2, 3), 5)`	It works, but when it fails the message reads backwards: `expected` shows the actual result	Expected first, then actual
Forgetting the tolerance for floating-point numbers: `assertEquals(0.3, 0.1 + 0.2)`	`expected: <0.3> but was: <0.30000000000000004>`	`assertEquals(0.3, 0.1 + 0.2, 1e-9)`
Missing `import static`	Compile error: `cannot find symbol` , `method assertEquals(int,int)`	`import static org.junit.jupiter.api.Assertions.*;`
Mixing JUnit 4 and 5	`org.junit.Test` and `org.junit.jupiter.api.Test` are different; a wrong import can make the tests not run at all	In a JUnit 5 project, only `org.junit.jupiter.api`
A `@BeforeAll` method that is not `static`	JUnit does not run the tests in the class: `@BeforeAll method '…' must be static …`	`static void start()`
Dependencies between tests	When the order changes, the tests fail randomly	Each test sets up its own data
A test without an assertion	The test is always ✓ and proves nothing; it still raises coverage	At least one assertion in every test
Testing only the "happy path"	Boundaries (0, empty array, negative), invalid inputs, and exceptions are forgotten; the bugs are exactly there	Normal case + boundary + invalid input
Comparing arrays with `assertEquals`	Fails even when the contents are the same; the references are compared	`assertArrayEquals`
An unseeded `Random` or `LocalDate.now()`	A flaky test: sometimes passes, sometimes fails	Supply the dependency from outside, use a fixed seed

Table 7.21: Common mistakes when writing tests.

Most of the mistakes in Table 7.21 get past the compiler. A reversed `assertEquals` , a test without an assertion, or two tests that depend on each other produce no warning at all. For this reason, test code must be read as carefully as production code. A simple way to see that a test itself is correct is to deliberately make it go red once: the expected value is written wrongly on purpose, and one checks that the test really fails. A test that never fails may not be testing anything.

7.10 Summary

- Testing systematically checks whether a program shows the expected behavior. A test can show the presence of bugs but not their absence; well-chosen tests raise confidence.
- An error is the programmer's mistake, a defect is its counterpart in the code, and a failure is the wrong behavior seen at run time. Testing exposes the failure, and debugging finds the defect.
- The test levels are unit, module, sub-system, system, and acceptance testing. A bug caught low is cheap; one caught high is expensive. Black-box testing starts from the requirements, white-box testing from the code.
- In JUnit 5, test methods are marked with `@Test`; the result is checked with assertions such as `assertEquals(expected, actual)`. Use a tolerance for floating-point numbers and `assertArrayEquals` for arrays.
- A good test follows the arrange, act, assert (AAA) pattern and checks a single behavior. The test name describes the behavior.
- JUnit creates a new test object for every test. `@BeforeEach` runs before every test and `@AfterEach` after every test (even if the test fails). `@BeforeAll` and `@AfterAll` run once and must be `static`.
- `assertThrows` verifies that the code throws an exception of the expected type (or a subtype) and returns the exception object.
- Bugs occur most often at the boundaries. Boundary value analysis chooses both ends of each range and the first values of the invalid region; a single representative of an equivalence class is enough. `@ParameterizedTest` and `@CsvSource` run the same test with many data rows.
- A good unit test is fast, independent, repeatable, self-validating, and timely (FIRST). External dependencies are replaced with test doubles.
- Regression testing means rerunning all the tests after every change. Tests must be deterministic; randomness is controlled with a fixed seed and dependency injection. CI systems run the tests on every push.
- In TDD, a failing test is written first (red), then the simplest code that makes it pass (green), and then the code is cleaned up (refactoring).
- The debugger stops the program at a breakpoint, before the line runs. Step Into enters a method, Step Over passes a call in a single step, and Step Out finishes the current method. Conditional and exception breakpoints catch the moment you are looking for. `jdb` does the same job from the command line.
- Systematic debugging: reproduce, write a failing test, narrow down, form and test a hypothesis, fix and run all the tests. `assert` assumptions are checked when `-ea` is on.
- A profiler (JFR, VisualVM) measures the hot spot where time is spent: measure first, then optimize. Code coverage (JaCoCo) shows lines and branches that never run; 100% coverage does not mean defect-free code.

- Static analysis (`javac -Xlint`, SpotBugs) finds suspicious patterns without running the code. Property-based testing checks a property with random inputs; if the property is weak, it also approves faulty code.
- Selenium tests a web application in a real browser. A memory leak is a reference that remains to objects that are no longer needed; it shows up as rising troughs in VisualVM, and the heap dump is examined with Eclipse Memory Analyzer.

7.11 Key terms

testing error / defect / failure debugging debugger unit test

module, sub-system, system, acceptance testing black-box / white-box testing framework

annotation assertion lambda expression test lifecycle arrange, act, assert boundary value analysis

equivalence class parameterized test integration test test double regression test flaky test seed

dependency injection continuous integration test-driven development breakpoint call stack

stack frame profiler hot spot code coverage static analysis property-based testing

counterexample garbage collector memory leak heap (memory) heap dump

---

7.12 Test Yourself

In the questions, the classes used in the note (`BankAccount`, `StepDemo`, `Jdb`, `Die`) apply exactly as written.

1. Does `assertEquals(0.1 * 3, 0.3)` pass? If it fails, what is the message, and what is misleading about it?

   Answer

   It fails: `expected: <0.30000000000000004> but was: <0.3>`. There are two problems. There is no tolerance: 0.1 × 3 is not exactly 0.3 in binary. The parameters are also in the wrong order: the computed value has taken the place of "expected," and the constant 0.3 looks like the "actual" value. The correct form is `assertEquals(0.3, 0.1 * 3, 1e-9)`.

2. A test class has three ordinary `@Test` methods and a `@ParameterizedTest` whose `@CsvSource` has three rows. How many times does the `@BeforeEach` method run?

   Answer

   6 times. Each row of the parameterized test is a separate test invocation, and `@BeforeEach` runs before every invocation: 3 + 3 = 6.

3. For an account with a balance of 100, does `assertThrows(IllegalStateException.class, () -> acct.deposit(-10))` pass?

   Answer

   It fails. `deposit(-10)` throws an exception, but its type is `IllegalArgumentException`; this class is not a subclass of `IllegalStateException`. The message: `Unexpected exception type thrown, expected: <java.lang.IllegalStateException> but was: <java.lang.IllegalArgumentException>`.

4. In the `Jdb` program, a breakpoint is set on line 3 (`int result = x * x;`) with `stop at Jdb:3`. How many times does the program stop? What does `locals` show at the first stop?

   Answer

   It stops three times: `square` is called three times in the loop, and `x` is 1, 2, 3 in turn. At the first stop, `locals` shows only the method argument `x = 1`; the list of local variables is empty. Since line 3 has not run yet, `result` does not exist at that moment.

5. The method `static String kind(int n)` consists of three executable lines: `if (n % 2 == 0)`, `return "even";`, and `return "odd";`. If there is only a test for `kind(4)`, what are the line and branch coverage?

   Answer

   4 is even; the `if` line and `return "even";` run. Line coverage is 2 / 3 ≈ 67%. Of the two branches of the single condition, only the true branch has been tried: branch coverage is 1 / 2 = 50%. A second test with an odd input, for example `kind(3)`, brings both to 100%.

6. In the `StepDemo` program, the debugger is stopped inside `square` with `x = 3`. First Step Out is made, then Step Over. Which line is the program on, and what is `sq1`?

   Answer

   Step Out finishes `square(3)` and returns to line 7, `int sq1 = square(a);`; the return value is 9, but the assignment has not completed yet. Step Over completes the assignment and moves to line 8. `sq1 = 9`.

7. A store gives a 10% discount to anyone who buys 6 or more items; the number of items cannot be

negative. Which inputs does boundary value analysis choose?

Answer

Equivalence classes: invalid (< 0), no discount (0–5), discount (≥ 6). Boundaries: −1 (the first value of the invalid region), 0 (the start of the valid region), and 5 and 6 (the pair that encloses the discount threshold from both sides). If the threshold had mistakenly been written as `qty > 6` instead of `qty >= 6`, only 6 would catch this defect.

8. Suppose `TextUtils.reverse` had mistakenly been written as `return s;`: it does not change the string at all. Does the property test "reversing twice does not change the string" catch this defect? How can it be strengthened?

Answer

It does not: `reverse(reverse(s))` is still `s`, and all 1000 inputs pass. The property is correct, but on its own it is weak. To strengthen it, add a second property: the first character of the result is the last character of the input. Or verify a few known examples: `assertEquals("cba", TextUtils.reverse("abc"))`.

7.13 Further reading

- JUnit Team, *JUnit 5 User Guide* (junit.org/junit5/docs/current/user-guide), the sections "Writing Tests", "Assertions", and "Parameterized Tests": the official and up-to-date account of annotations, assertions, and parameter sources.
- K. Beck, *Test-Driven Development: By Example*: applies the red-green-refactor cycle step by step on a currency example; it is the book of the person who established the method.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals* (12th ed.), Chapter 7, "Exceptions, Assertions, and Logging": covers the `assert` statement, logging, and debugging tips with Java-specific details.
- Oracle, *Programming With Assertions*: a short official document that explains when to use `assert` and when not to.

7.14 References

1. JUnit Team, *JUnit 5 User Guide*, junit.org/junit5/docs/current/user-guide (Writing Tests, Assertions, Parameterized Tests).
2. Oracle, *The Java Tutorials* and *JDK 21 Tool Specifications*: `javac` (`-Xlint`), `jdb`, `jcmd`, `jfr`.
3. Oracle, *Programming With Assertions* (Java SE documentation).
4. J. Gosling et al., *The Java Language Specification, Java SE 21 Edition* (14.10 The assert Statement).
5. P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson.

6. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson (Chapter 7: Exceptions, Assertions, and Logging).
7. K. Beck, *Test-Driven Development: By Example*, Addison-Wesley.
8. Tool documentation: SpotBugs (spotbugs.github.io), JaCoCo (jacoco.org), Selenium (selenium.dev), Eclipse Memory Analyzer (eclipse.dev/mat), VisualVM (visualvm.github.io), jqwik (jqwik.net).

PART IV

Applications in Java

Chapters 8–11

PART IV · CHAPTER 8

Input and Output

A program's variables live in memory and vanish when the program ends. Everything that has to last, such as measurement logs, configuration files, images, and student lists, is kept somewhere outside the program: on a disk, across a network, or on a device. A program's exchange of data with this outside world is called input and output (I/O). Java builds all of it on a single idea: data is a stream that flows in order from a source to a destination. These notes first introduce the stream concept and the standard streams of the keyboard and the screen. They then go down to the bytes inside files: the difference between text and binary files, character encodings, and how many bytes a number occupies in a file. The following sections cover the `File` class, the classes for reading and writing text, buffering, the other byte streams, and the ways of copying a file. The last sections treat random access, object serialization, and NIO, Java's library built on buffers and channels.

LEARNING OUTCOMES

Explain streams
Describe input and output streams and the `System.in`, `System.out`, and `System.err` streams; read from the keyboard with `Scanner`.

Tell text from binary
Explain the difference between text and binary files and the ASCII, ISO-8859-9, and UTF-8 encodings; compute how many bytes a number occupies in a file.

Read and write files
Process text files with `File`, `Scanner`, `BufferedReader`, `PrintWriter`, and `FileWriter`; close resources with try-with-resources.

Choose the right stream
Distinguish byte streams (`InputStream` / `OutputStream`) from character streams (`Reader` / `Writer`), and buffered streams from wrapper streams.

Copy and seek
Copy at the byte, block, and line level; jump to a given position with `RandomAccessFile`.

Store objects and use NIO
Serialize objects (`Serializable` , `transient`); explain what an NIO buffer, channel, and selector are for.

8.1 Streams

A program takes data from the outside, processes it, and hands the result back. The outside can be a different place each time: the keyboard, the screen, a file on disk, a server on the network, or an array in memory. Learning a separate way of reading for each of these sources would be hard and unnecessary. Java treats them all through one abstraction: the stream.

8.1.1 What is a stream?

Definition 8.1

A **stream** is a structure that carries data from a source to a destination in order, piece by piece. A stream through which data enters the program is an **input stream**, and a stream through which data leaves the program is an **output stream**. The pieces carried can be characters, numbers, or raw bytes.

A water pipe is a fitting analogy. Water that enters at one end of the pipe comes out at the other end in the same order, and whoever drinks the water does not care whether the far end is a tap or a tank. A stream works the same way. The program sees only its own end of the stream. The other end may be a keyboard, a file, or a network connection, and in every case the program calls the same `read()` and `write()` methods. The power of Java's I/O (input/output) library comes from this common interface. Figure 8.1 shows the structure.

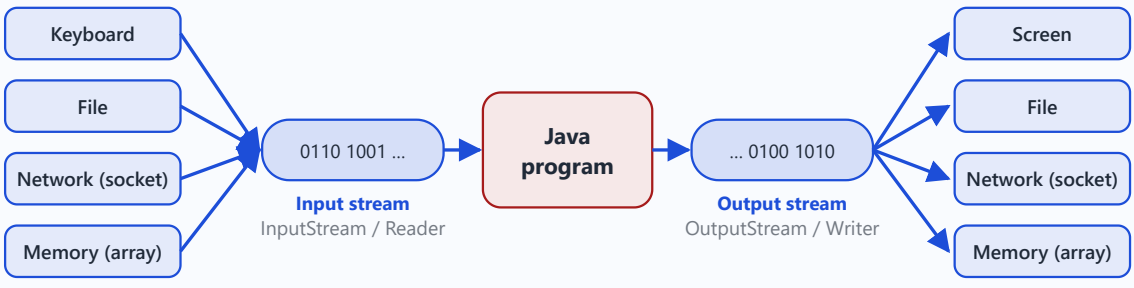


Figure 8.1: A stream. Data travels in order from the sources on the left to the program through an input stream, and from the program to the destinations on the right through an output stream. The superclasses of byte streams are `InputStream` and `OutputStream` ; those of character streams are `Reader` and `Writer` .

Two properties of streams matter throughout this chapter. The first is **order**: data read from a stream arrives in the order it was written, and an item cannot be skipped to reach the next one. For this reason a stream cannot jump straight to a record in the middle of a file; that requires random access. The second is **direction**: a stream is either an input stream or an output stream, never both. A program that wants to both read and write the same file opens two separate streams.

8.1.2 Standard streams

The keyboard and the screen are the simplest I/O devices. Data handled through them is transient: what is typed at the keyboard and written to the screen is not stored, and it disappears when the program ends. It is also immediate: input arrives as the user types, and output appears on the screen as the program writes it. When a Java program starts, it receives three ready-made streams from the operating system. They are called the **standard streams** (Table 8.1).

Stream	Java object	Type	Default endpoint
standard input	`System.in`	`InputStream`	keyboard
standard output	`System.out`	`PrintStream`	screen
standard error	`System.err`	`PrintStream`	screen

Table 8.1: The three standard streams of a Java program.

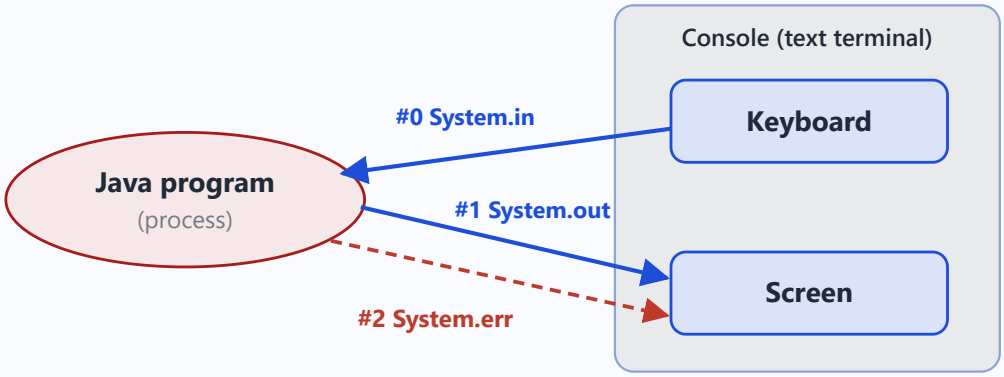


Figure 8.2: The standard streams. The keyboard is connected to the program through `System.in`; the program writes to the screen through `System.out` and `System.err`. The numbers are the operating system's file descriptor numbers (0, 1, 2).

`System.out` and `System.err` both write to the screen. So why are there two streams? The difference shows when output is redirected. If you type `java Application > result.txt` on the command line, the `>` sign redirects only standard output to the file `result.txt`. Whatever is written to `System.err` stays on the screen. This way error messages neither pollute the output file nor go unnoticed. A program's actual result goes to `System.out`; warnings and error messages go to `System.err`.

The type of `System.in` is `InputStream`, so it delivers raw bytes. When a user types `42` at the keyboard and presses Enter, three bytes enter the stream: `'4'`, `'2'`, and the line terminator. Converting these bytes into an `int` is left to the programmer. The class `java.util.Scanner` does that job.

8.1.3 A first example: reading and writing

The following program uses all three standard streams. It reads a number from the keyboard, writes two lines to standard output, and writes one line to standard error.

```
import java.util.Scanner;
public class StreamDemo {
  public static void main(String[] args) {
    // Read from the input stream
    Scanner scanner = new Scanner(System.in);
    int userInput = scanner.nextInt();
    // Write to the output stream
    System.out.println("Hello, World!");
    System.out.println("You entered: " + userInput);
    System.err.println("This line went to stderr.");
  }
}
```

If the user types `42`, the console shows the following lines (the first line is the input the user typed):

```
42
Hello, World!
You entered: 42
This line went to stderr.
```

The program runs line by line as follows:

- `new Scanner(System.in)` creates a scanner that reads the keyboard stream. `Scanner` is in the `java.util` package, which is why the `import` line at the top of the file is required.
- `scanner.nextInt()` waits until the user types a number and presses Enter. This waiting is called **blocking**: the program stops at that line and continues once input arrives.
- `System.out.println` writes a line to the output stream and ends the line.
- `System.err.println` writes to the error stream. Most development environments show this line in red.

If the user types `abc` instead of a number, `nextInt()` cannot convert the text to a number and throws `java.util.InputMismatchException`. The program ends with an uncaught exception:

```
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:947)
        at java.base/java.util.Scanner.next(Scanner.java:1602)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2267)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2221)
        at StreamDemo.main(StreamDemo.java:7)
```

This is an unchecked exception; the compiler does not force you to catch it. In programs that cannot trust user input, check first with `hasNextInt()`, or catch the exception and ask the user again.

8.1.4 Code trace: reading from the keyboard

The following program asks for the user's name and age and computes the age next year. It uses `nextLine()` to read the name and `nextInt()` to read the age. The two methods treat the line terminator differently, which causes one of the most common mistakes in this chapter; the trace makes the difference visible.

```
import java.util.Scanner;
public class ConsoleInput {
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Name: ");
    String name = keyboard.nextLine();
    System.out.print("Age: ");
    int age = keyboard.nextInt();
    int nextYear = age + 1;
    System.out.println("Hello " + name + "!");
    System.out.println("Next year you'll be " + nextYear + ".");
    keyboard.close();
  }
}
```

Suppose the user types `Ayşe Yılmaz` as the name and `19` as the age. Table 8.2 gives every step of the program. The line numbers refer to the lines of the code above.

Step	Line	Variables	New output
1	5: `Scanner keyboard = new Scanner(System.in);`	keyboard=Scanner(System.in)	—
2	6: `System.out.print("Name: ");`	keyboard=Scanner(System.in)	`Name:`
3	7: `String name = keyboard.nextLine();`	name="Ayşe Yılmaz"	`Ayşe Yılmaz`
4	8: `System.out.print("Age: ");`	name="Ayşe Yılmaz"	`Age:`
5	9: `int age = keyboard.nextInt();`	name="Ayşe Yılmaz", age=19	`19`
6	10: `int nextYear = age + 1;`	name="Ayşe Yılmaz", age=19, nextYear=20	—

Step	Line	Variables	New output
7	11: `System.out.println("Hello " + name + "!");`	name="Ayşe Yılmaz", age=19, nextYear=20	`Hello Ayşe Yılmaz!`
8	12: `System.out.println("Next year you'll be " + nextYear + ".");`	name="Ayşe Yılmaz", age=19, nextYear=20	`Next year you'll be 20.`
9	13: `keyboard.close();`	name="Ayşe Yılmaz", age=19, nextYear=20	—

Table 8.2: Trace of the `ConsoleInput` program. The "New output" column shows the text added to the console in that step; ⏎ is the line terminator. The console also shows what the user typed, `Ayşe Yılmaz` and `19`.

In the first step the `Scanner` object is attached to the `System.in` stream. Nothing has been read yet, and the keyboard buffer is empty. In the second step `print` (not `println`) writes the prompt `Name:` . Because `print` does not end the line, the user's answer is typed right next to the prompt.

The third step is the important one. The user types `Ayşe Yılmaz` and presses Enter. The keyboard buffer now holds `Ayşe Yılmaz⏎` . `nextLine()` takes *everything* up to the line terminator, spaces included, and also discards the terminator from the buffer. The variable `name` is assigned `"Ayşe Yılmaz"` ; the line terminator is not part of this text.

In the fifth step the user types `19` and presses Enter. `nextInt()` skips leading whitespace, reads the digits, and converts them to the `int` value 19. It does not touch the line terminator: `⏎` stays in the buffer. This causes no trouble here, because no further read follows. In the later steps `age` is an ordinary `int` : `19 + 1 = 20` . The output lines are built by string concatenation: `"Next year you'll be " + 20 + "."` .

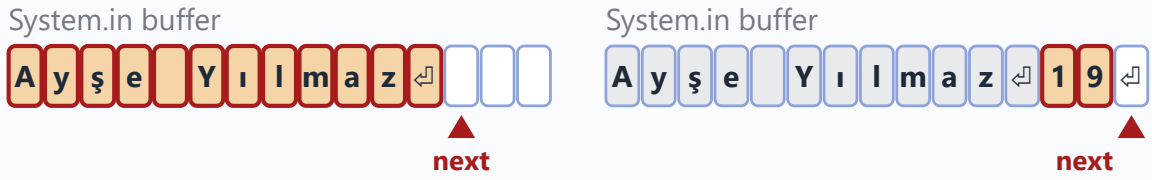


Figure 8.3: Two moments of the keyboard buffer. (a) `nextLine()` takes the whole line and discards the line terminator. (b) `nextInt()` takes the digits; the line terminator stays in the buffer.

The `keyboard.close()` on the last line closes not just the scanner but also the `System.in` stream it wraps. A closed standard input cannot be reopened. Even if the program later creates a new `Scanner(System.in)` , it cannot read from the keyboard. The small experiment below shows this: after the first scanner is closed, the second one throws `NoSuchElementException` even though a line is waiting in the input.

```
read: one
java.util.NoSuchElementException: No line found
```

For this reason, the scanner that reads the keyboard is created once in a program and, if at all, closed only at the very end. Scanners that read files, on the other hand, should be closed as soon as the work is done; the file sections return to this difference.

8.1.5 `nextLine` after `nextInt`

The effect of the leftover line terminator shows up when the order is reversed. Suppose the age is read first and the name second:

```
import java.util.Scanner;
public class LineAfterInt {
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Age: ");
    int age = keyboard.nextInt();
    System.out.print("Name: ");
    String name = keyboard.nextLine();
    System.out.println("[" + name + "] " + age);
  }
}
```

The user types `19` and presses Enter. `nextInt()` reads 19 and leaves the line terminator in the buffer. Then `nextLine()` is called. This method asks for the text up to the next line terminator. A line terminator is already waiting in the buffer, and the text in front of it is empty. `nextLine()` returns the empty string without waiting at all. The program ends before the user gets a chance to type a name:

```
Age: Name: [] 19
```

Common mistake

Reading text with `nextLine()` right after a call to `nextInt()`, `nextDouble()`, or `next()`. The first `nextLine()` reads the line terminator left over from the previous line and returns an empty string. There are two fixes: insert an extra `keyboard.nextLine();` that discards the terminator, or read the number as a line too and convert it: `int age = Integer.parseInt(keyboard.nextLine());`.

The second fix is usually cleaner. Because every read consumes a whole line, nothing is left in the buffer. If the user types extra spaces next to the number, `parseInt` fails; in that case use `keyboard.nextLine().trim()`.

8.2 Files and encodings

Exchange through the keyboard and the screen is transient. Lasting data needs a file. This section explains what a file really is, how text files differ from binary files, and how characters are converted into bytes. This knowledge decides which stream class to choose in the later sections.

8.2.1 A file: a sequence of bytes on disk

A file stores data permanently on disk. The data stays in place even when the program ends or the computer restarts. Variables in RAM, by contrast, vanish with the program. The data in every file ultimately consists of 0s and 1s, that is, of a **sequence of bytes**. What distinguishes one file from another is how these bytes are interpreted.

> **Definition 8.2**
>
> In a **text file**, bytes are converted into letters according to a character encoding; a person can read the file with a text editor. In a **binary file**, the bytes are data such as numbers, pixels, audio samples, or machine instructions; only a program that knows the format of the file can make sense of them.

To see the bytes of a file, you use a hexadecimal (hex) editor. This tool shows each byte as two hexadecimal digits. A file `file.txt` containing `Merhaba, dünya!` looks like this in a hex editor:

```
0000  4D 65 72 68 61 62 61 2C  Merhaba,
0008  20 64 C3 BC 6E 79 61 21   d..nya!
```

The left column is the position of the first byte on each line: `0000` and `0008`. The middle holds the hexadecimal value of each byte, and the right the ASCII equivalent. The first byte is `4D`: $4 \cdot 16 + 13 = 77$, which is the capital letter `M` in ASCII. The text has 15 characters, but the file takes 16 bytes. The difference comes from the letter `ü`: in UTF-8, `ü` is two bytes, `C3 BC`. The two dots in the right column show that these two bytes have no ASCII equivalent. If the file is produced with Java and its bytes are printed, the same 16 values appear:

```
16 bytes: 4D 65 72 68 61 62 61 2C 20 64 C3 BC 6E 79 61 21
1 byte: 7F
1382 bytes: CA FE BA BE 00 00 00 41 00 5C 0A 00 02 00 03 07 00 04 0C 00 05 00 06 01 00 10
6A 61 76 61 2F 6C
·······A·\··············java/lang/Object···<init>···()V·····java/io/FileWriter·····fil
e.txt·············(Ljava/lang
major = 65
```

The first line of the output gives the bytes of this file, and the second line gives the bytes of the file `data.dat`. The remaining lines show the beginning of a `.class` file; these are explained when binary files are discussed.

8.2.2 The file-processing cycle

Whatever language is used, working with a file goes through the same steps. Figure 8.4 shows the steps, and Table 8.3 gives the Java counterpart of each.

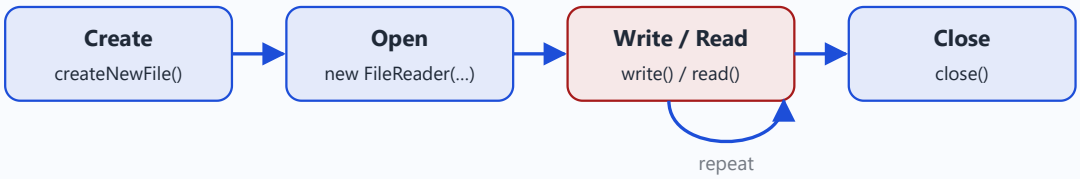


Figure 8.4: The file-processing cycle. The file is created or opened, read or written, and finally closed. The read-and-write step can be repeated as often as needed.

Step	What happens	In Java
Create	A new, empty file appears on disk.	`file.createNewFile()`; opening a file for writing also creates it
Open	A stream is attached to an existing file.	`new FileInputStream(name)`, `new FileReader(name)`, `new Scanner(file)`
Write	Data is transferred to the file.	`write()`, `print()`, `println()`
Read	Data is brought from the file into the program.	`read()`, `readLine()`, `nextLine()`
Close	Data left in the buffer is written to disk, and the operating system releases the resource.	`close()` or try-with-resources

Table 8.3: The file-processing steps and their counterparts in Java.

The last step is the one most often forgotten. A file that is not closed causes two problems. The first is **data loss**: some of the bytes written wait in a buffer in memory and may never reach the disk unless the file is closed. The second is a **resource leak**: the operating system allocates a file handle for every open file, and the number of files a process can keep open at once is limited. A long-running program that never closes its files eventually cannot open new ones. On Windows, a file left open also cannot be deleted or moved.

8.2.3 A first file program

The following program writes one sentence to a text file and a single byte to a binary file. It then reads the byte back from the binary file and prints the size of the text file.

```
import java.io.*;

public class FirstFile {
  public static void main(String[] args) throws IOException {
    // Write to a text file
    FileWriter writer = new FileWriter("file.txt");
    writer.write("Merhaba, dünya!");
    writer.close();
    // Write one byte to a binary file
    FileOutputStream out = new FileOutputStream("data.dat");
    out.write(127);
    out.close();
    // Read from the binary file
    FileInputStream stream = new FileInputStream("data.dat");
    int b = stream.read();
    stream.close();
    System.out.println("Byte read: " + b);
    File text = new File("file.txt");
    System.out.println("file.txt: " + text.length() + " bytes");
  }
}
```

```
Byte read: 127
file.txt: 16 bytes
```

After the program has run, the contents of `file.txt` are:

```
Merhaba, dünya!
```

The program uses three different stream classes:

- `FileWriter` writes **text**. It converts the characters it is given into bytes with a character encoding and sends them to the file. Since Java 18 the default encoding is UTF-8; that is why `ü` becomes two bytes and the 15-character text takes 16 bytes.
- `FileOutputStream` writes **raw bytes**. The call `write(127)` puts a single byte in the file: `7F` in hexadecimal, `0111 1111` in binary. No character conversion takes place.
- `FileInputStream` reads raw bytes. `read()` returns a byte as an `int` between 0 and 255. If there is no byte left to read, it returns −1; this detail matters in the section on copying.

The `throws IOException` in the header of `main` declares that file operations can throw a checked exception. The file may not exist, the disk may be full, or permission may be missing. The compiler requires these cases to be handled: either caught with `try`-`catch` or declared with `throws` in the method header. These short examples choose the second way. In real programs it is better to catch the exception so that the error message can be given to the user in an understandable form.

8.2.4 Text files and character encodings

Text files contain printable characters: letters, digits, punctuation marks, spaces, and line terminators. They can be opened and edited with text editors such as Notepad or VS Code. Source code (`.java` , `.c`), `.txt` , `.csv` , and `.html` files are text files.

A computer cannot store letters, only numbers. So every character is assigned a number. The rule of this mapping is called a character encoding.

Definition 8.3

A **character encoding**, also called a character set, is the rule that determines with which number, that is, with which byte or bytes, each character is stored. The programs that write and read a text file must use the same encoding.

Many encodings have appeared over the course of history. The four met in this course are summarized in Table 8.4.

Encoding	Per character	Coverage
ASCII	7 bits (1 byte)	128 characters: English letters, digits, punctuation, and control characters.
ISO-8859-1 (Latin-1)	1 byte	256 characters, Western European languages. It has `é` , `ü` , `ö` , `ç` ; it does not have `ş` , `ğ` , `ı` .
ISO-8859-9 (Latin-5)	1 byte	The Turkish variant of Latin-1. Six letters of Latin-1 are replaced by `ğ Ğ ı İ ş Ş` .
UTF-8	1–4 bytes	The universal encoding of Unicode; all the world's languages. The default encoding of the web and, since Java 18, of Java.

Table 8.4: Commonly used character encodings.

UTF-8 is a variable-length encoding. The 128 ASCII characters are written in UTF-8 with the same single byte, so an ASCII file containing only English letters is also a valid UTF-8 file. Turkish letters and other Latin letters take two bytes, symbols such as the euro sign `€` take three, and emoji take four. The experiment below measures these numbers with the `getBytes` method:

```
32 characters, UTF-8 41, ISO-8859-9 32, UTF-16BE 64
IÅ<9f>Ä±k, Å<9f>imÅ<9f>ek, Ã§iÄ<9f>, Ã¶Ä<9f>renci, Ã<9c>nye
ş: ASCII 3F, UTF-8 C5 9F, ISO-8859-9 FE, cp1254 FE
þimdi
şimdi
Çağrı UTF-8: 8, ISO-8859-9: 5, UTF-16BE: 10
€ UTF-8: 3, 😀 UTF-8: 4, 😀 length(): 2
default: UTF-8 / native.encoding=Cp1254
4D 65 72 68 61 62 61 2C 20 64 C3 BC 6E 79 61 21
```

The first line shows that a 32-character Turkish sentence takes 41 bytes in UTF-8, 32 in ISO-8859-9, and 64 in UTF-16. The sentence has nine Turkish letters that take two bytes each: 32 + 9 = 41. The second line is the same bytes decoded with the wrong encoding; this garbling is discussed in detail in the section on specifying the encoding. The `ş` line gives the bytes of one letter in four encodings. ASCII has no `ş`; Java writes a question mark (`3F`) in its place. The "Çağrı" line gives an interesting detail too: `"Çağrı"` has five characters but takes 8 bytes in UTF-8. `Ç`, `ğ`, and `ı` take two bytes each, and `a` and `r` one each: 2 + 1 + 2 + 1 + 2 = 8. The same text is 5 bytes in ISO-8859-9 and 10 bytes in UTF-16. For 😜, `length()` returns 2. Java's `char` type is 16 bits wide, and characters with code points above 65,535 are represented by two `char` values. So the length of a string does not always equal either the number of characters or the number of bytes.

The "default" line in the output deserves attention too. In Java 18 and later, the default encoding is UTF-8 regardless of the operating system. The operating system's own encoding is stored in the `native.encoding` property; on Turkish Windows this is Windows-1254 (Cp1254). In Java 17 and earlier, the default encoding was directly this operating system setting. The same program could write different bytes on Windows and on Linux.

8.2.5 Binary files

Binary files store bytes not as characters but in a format chosen by the program: numbers, pixels, audio samples, machine code. These bytes are meaningful only to a program that knows the format; a person cannot read them directly. Executable files (`.exe`, Java's `.class` files), images (`.png`, `.jpg`), music (`.mp3`), video (`.mp4`), and data files (`.dat`) are binary files. The advantages of binary files are that they are smaller and faster to process: a number does not have to be converted to text and back.

The first 32 bytes of `FirstFile.class`, produced by compiling the previous program, look like this in a hex editor:

```
0000  CA FE BA BE 00 00 00 41  .......A
0008  00 5C 0A 00 02 00 03 07  .\......
0010  00 04 0C 00 05 00 06 01  ........
0018  00 10 6A 61 76 61 2F 6C  ..java/l
```

Every `.class` file begins with the bytes `CA FE BA BE`. These four bytes are called the **magic number**; the JVM recognizes from it that the file really is a class file. The next four bytes are version information. `00 41` = 65 is the class-file version of Java 21; this number was also read from a file compiled with JDK 21, using Java itself (`major = 65` in the earlier output). The constant pool follows. Some readable pieces are visible as well: class names such as `java/lang/Object` are stored in the file as text. When the same file is opened in Notepad, a meaningless string of characters appears:

```
Êþº¾···A·\···············java/lang/Object···<init>···()V·····java/io/FileWriter·····fil
e.txt·············(
```

Notepad tries to decode the bytes with a text encoding. `CA`, `FE`, `BA`, and `BE` are the characters `Ê`, `þ`, `º`, and `¾` in Latin-1. Bytes that cannot be printed appear as dots or spaces. Opening a binary file in a text editor and saving it can corrupt it: the editor may change some bytes or convert the line terminators.

8.2.6 Extended ASCII and Turkish characters

ASCII is 7 bits and covers 128 characters. A byte, however, is 8 bits. The one remaining bit makes room for 128 new characters.

Definition 8.4

Extended ASCII is the general name for encodings that use all 8 bits to represent $2^8 = 256$ characters. Codes 0–127 are the standard ASCII characters and are the same in every table. Codes 128–255 are reserved for special characters, symbols, and national letters.

There is no single "extended ASCII". Which characters the upper half holds varies with the **code page**. Two code pages are used for Turkish: ISO-8859-9 (Latin-5) and Windows-1254. They use the same bytes for the Turkish letters. Table 8.5 shows the meaning of the same bytes in Latin-1 and Latin-5 side by side.

Byte	D0	DD	DE	F0	FD	FE	FC	E7
ISO-8859-1	Ð	Ý	Þ	ð	ý	þ	ü	ç
ISO-8859-9	Ğ	İ	Ş	ğ	ı	ş	ü	ç

Table 8.5: The meaning of the same byte in two code pages.

`FC` and `E7` are `ü` and `ç` in both tables; these letters exist in Western European languages too. The first six bytes, however, change from table to table. The same byte `FE` is `ş` in one table and the Icelandic letter `þ` in the other. When a Turkish file is opened with the wrong code page, garbled text such as “þimdi” instead of “şimdi” appears (Figure 8.5). The `þimdi` and `şimdi` lines in the earlier output were obtained from the same five bytes with the two code pages.



Figure 8.5: The byte `FE` is `ş` in ISO-8859-9 and `þ` in ISO-8859-1. Opening a file with the wrong code page garbles the text.

The lasting solution to this problem is Unicode, which collects all languages in a single table, and its byte encoding UTF-8. In UTF-8 each character has exactly one byte sequence; the reader does not have to guess which code page to choose. UTF-8 is the choice for every text file written today.

8.2.7 Writing 127 to a file

How a number is stored in a file depends on its data type, its purpose, and the file format. The number 127 can be written in two fundamentally different ways.

- Written **as text**, the number is first converted into characters: `"127"`. Each digit takes one ASCII byte. A point that is often confused matters here: the code of the character `'1'` is not 1 but 49. `'2'` is 50 and `'7'` is 55. The file holds the bytes `31 32 37`.
- Written **as binary**, the binary value of the number, `0111 1111`, is written directly into as many bytes as the type is wide: 1 byte for `byte`, 2 for `short`, and 4 for `int`. For numeric data this way is more efficient.

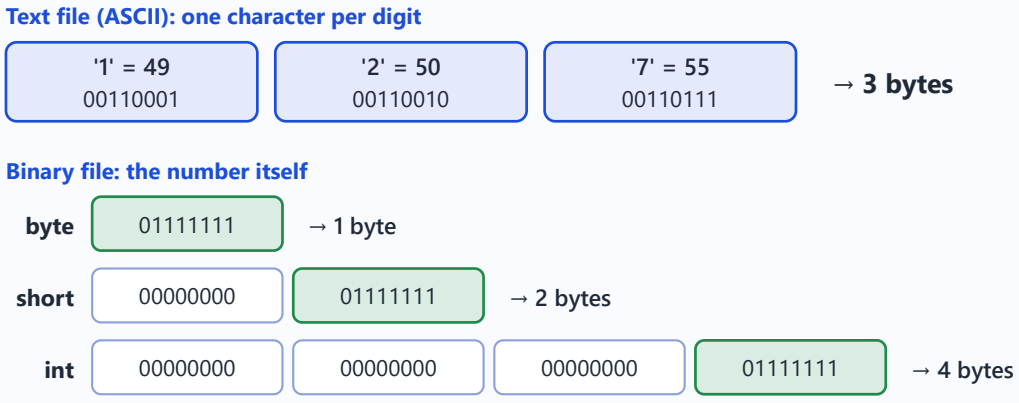


Figure 8.6: Representations of the number 127 in a file. In a text file each digit is one ASCII byte, 3 bytes in all. In a binary file the number itself is written into as many bytes as the type is wide: 1, 2, or 4 bytes.

The following program really writes these five representations to files and measures the file sizes. It uses `OutputStreamWriter` to write text and `DataOutputStream` to write binary. For brevity, the lines `import java.io.*;` and `import java.nio.charset.*;` are not shown.

```
public class Number127 {
  static void writeText(String name, Charset cs) throws IOException {
    Writer w = new OutputStreamWriter(new FileOutputStream(name), cs);
    w.write(String.valueOf(127));            // "127": 3 chars
    w.close();
  }
  static void writeBinary(String name, int size) throws IOException {
    FileOutputStream fos = new FileOutputStream(name);
    DataOutputStream d = new DataOutputStream(fos);
    if (size == 1) d.writeByte(127);        // 01111111
    else if (size == 2) d.writeShort(127);  // 00000000 01111111
    else d.writeInt(127);                    // 4 bytes
    d.close();
  }
  public static void main(String[] args) throws IOException {
    writeText("text.txt", StandardCharsets.US_ASCII);
    writeText("text16.txt", StandardCharsets.UTF_16BE);
    writeBinary("b.dat", 1);  writeBinary("s.dat", 2);  writeBinary("i.dat", 4);
    String[] names = {"text.txt", "text16.txt",
             "b.dat", "s.dat", "i.dat"};
    for (String name : names)
      System.out.println(name + ": " + new File(name).length());
  }
}
```

```
text.txt: 3
text16.txt: 6
b.dat: 1
s.dat: 2
i.dat: 4
```

The `writeText` method wraps a `FileOutputStream` in an `OutputStreamWriter`. `OutputStreamWriter` is the bridge class that converts characters into bytes with the encoding it is given. With ASCII, `"127"` takes three bytes. `UTF_16BE` writes each character with two bytes and takes 6 bytes. The BE (big-endian) in the name of this encoding fixes the byte order, so no byte order mark (BOM) is written at the start of the file. Had the `UTF_16` encoding with a byte order mark been used, two bytes `FE FF` would have been added at the start and the file would have taken 8 bytes.

The `writeBinary` method uses `DataOutputStream`. This class writes primitive types in their fixed-size binary representations: `writeByte` 1, `writeShort` 2, and `writeInt` 4 bytes. The bytes are written with the most significant byte first. The actual contents of the files confirm the measurement:

```
text.txt: 3
text16.txt: 6
b.dat: 1
s.dat: 2
i.dat: 4
text.txt -> 31 32 37
text16.txt -> 00 31 00 32 00 37
b.dat -> 7F
s.dat -> 00 7F
i.dat -> 00 00 00 7F
UTF_16 (with BOM): 8
two ints: 8
7 2000000000
wrong order/type: 0 7
EOFException: null
```

The last lines of the output show two important rules about `DataInputStream`. Values in a binary file are read back correctly only **in the order and with the type in which they were written**. If a file with two `int` values is read with two `readShort()` calls, 0 and 7 come back; that is, the four bytes of the first `int` are split into two pieces. If `readInt()` is called when the file has no more data, an `EOFException` (end of file) is thrown. A text file explains its own format to a person; for a binary file, the program that writes it and the program that reads it must know the format in advance.

8.2.8 Text or binary?

Table 8.6 compares the five representations of the number 127 by size.

How was 127 written?	Bytes	Size
ASCII / UTF-8 text file	`31 32 37`	3 bytes
UTF-16 text file (no BOM)	`00 31 00 32 00 37`	6 bytes
Binary, `byte`	`7F`	1 byte
Binary, `short`	`00 7F`	2 bytes
Binary, `int`	`00 00 00 7F`	4 bytes

Table 8.6: The size of the number 127 in a file.



Figure 8.7: The UTF-8 text representation of 127 (three digit bytes) and its four-byte binary representation written with `writeInt`.

In the table the binary representation always looks smaller, but that is true only for 127. In binary, the size depends on the **type**: an `int` is always 4 bytes, whether its value is 7 or 2,000,000,000. In text, the size depends on the **number of digits**: 7 takes one byte, while 2,000,000,000 takes ten. For small numbers text can be shorter; for large numbers binary clearly wins. Binary reading is also fast because it does not have to parse the number.

The choice depends on the purpose:

- **Text** is for data that people will read. It opens in every editor, moves easily between platforms, and can be repaired by eye when it is damaged. Log files, configuration files, and CSV tables are text.
- **Binary** is for storing numeric data efficiently. It is smaller and faster, but you must know the format of the program that wrote it. Image, audio, and database files are binary.

Java has clear tools for these two ways. `DataOutputStream` writes primitive types in binary (`writeByte`, `writeShort`, `writeInt`, and so on). `DataInputStream` reads them back in the same order. `OutputStreamWriter` converts characters into bytes with a given encoding.

8.3 The file system: the `File` class

Before reading the contents of a file, you often ask questions about the file itself. Does the file exist? Is it a file or a directory? How large is it? Is it readable? The `java.io.File` class answers these questions.

8.3.1 A `File` is a path

> **Definition 8.5**
>
> A `File` object represents a **path** on disk: the name and location of a file or directory. `File` does not read or write the contents of a file; that is the job of streams. `File` only accesses the file system to answer questions about the path and to perform operations such as creating, deleting, and listing files or directories.

The most important consequence of this distinction is that `new File("sample.txt")` creates nothing on disk. The constructor merely stores the path in an object. The object is created even if the file is not on disk, and no exception is thrown. For a nonexistent file, `exists()` returns `false` and `length()` returns 0. The exception for a missing file comes only when a stream is opened to read it; for example, `new FileReader(f)` throws `FileNotFoundException`.

```
false 0 missing.txt false
mkdir: false, mkdirs: true
second createNewFile: false
non-empty directory delete: false
lastModified(missing) = 0
1 a/b
```

The first line of this output gives the values of `exists()`, `length()`, `getName()`, and `isFile()` for the nonexistent `missing.txt`. The remaining lines show the subtle points of the methods explained below. Table 8.7 summarizes the commonly used methods.

Method	Returns
`exists()`	whether the path exists on disk (`boolean`)
`isFile()` / `isDirectory()`	whether it is a file / a directory
`canRead()`, `canWrite()`, `canExecute()`	read, write, and execute permission
`createNewFile()`	whether the file was created (`false` if it already exists)
`mkdir()` / `mkdirs()`	whether the directory was created (`mkdirs` also creates intermediate directories)
`delete()`	whether it was deleted (a non-empty directory is not deleted)
`list()` / `listFiles()`	the contents of a directory: `String[]` / `File[]`
`length()`	size in bytes (`long`)
`lastModified()`	time of last modification, in milliseconds since 1 January 1970
`getName()`, `getAbsolutePath()`	the name, the full path

Table 8.7: Commonly used methods of the `File` class.

Most of these methods do not throw an exception on failure; they return `false`. `mkdir()` fails if the intermediate directories do not exist; in the output above, for `a/b/c`, `mkdir` returns `false` while `mkdirs` returns `true`. `createNewFile()` returns `false` if the file already exists and leaves the existing file alone. `delete()` does not delete a non-empty directory and again returns `false`. The `lastModified()` of a nonexistent file is 0. Consequently, the return values of these methods must be checked. If the value is ignored, the operation fails silently and the error is noticed only much later.

8.3.2 File and directory operations

The following program creates a directory with an empty file inside it, queries the file's properties, and then deletes everything.

```
public class FileInfo {
  public static void main(String[] args) throws IOException {
    File dir = new File("sample");
    File file = new File("sample/file.txt");
    System.out.println("exists: " + file.exists());
    System.out.println("mkdir: " + dir.mkdir());
    System.out.println("created: " + file.createNewFile());
    System.out.println("isFile: " + file.isFile());
    System.out.println("isDir: " + file.isDirectory());
    System.out.println("readable: " + file.canRead());
    System.out.println("writable: " + file.canWrite());
    System.out.println("size: " + file.length());
    String[] items = dir.list();
    System.out.println("in dir: " + items.length + " item(s)");
    long time = file.lastModified();
    System.out.println("time > 0: " + (time > 0));
    System.out.println("deleted: " + file.delete());
    System.out.println("exists: " + file.exists());
    dir.delete();
  }
}
```

```
exists: false
mkdir: true
created: true
isFile: true
isDir: false
readable: true
writable: true
size: 0
in dir: 1 item(s)
time > 0: true
deleted: true
exists: false
```

On the first line the file does not exist yet. `mkdir()` creates the `sample` directory and `createNewFile()` creates the empty file inside it; both return `true`. Because the file was just created, its size is 0. `dir.list()` returns the names in the directory as an array; the length of the array is 1. The value of `lastModified()` differs every time the program runs, so the program prints only whether it is greater than zero. Finally the file is deleted and `exists()` is `false` again. The last statement deletes the directory; since the directory is now empty, the deletion succeeds.

Both paths in the program are **relative paths**: `"sample/file.txt"` is interpreted relative to the directory the program runs in. In a development environment this is usually the root directory of the project. `getAbsolutePath()` converts a relative path into a full path and shows where the file is really being looked for. When you get a "file not found" error, this is the first place to look.

The path separator is the backslash (`\`) on Windows and the slash (`/`) on Linux and macOS. Java accepts `/` on Windows too, so `"sample/file.txt"` works on every system. The system's own separator is stored in the constant `File.separator`. If backslashes are to be used in Windows paths, they must be written twice in a Java string: `"C:\\courses\\notes.txt"`. A single backslash starts an escape sequence. `"C:\courses"` does not compile, because `\c` is not a valid escape sequence:

```
Main.java:5: error: illegal escape character
    java.io.File f = new java.io.File("C:\courses\notes.txt");
                                        ^
1 error
```

The sneakier case is when the escape sequence is valid. `"C:\notes.txt"` compiles, because `\n` is a line terminator. The path silently turns into `C:` , a line terminator, and `otes.txt` :

```
COMPILED; ran: C:
otes.txt
```

8.4 Reading and writing text

Most file work is done with text files. This section explains the ways of reading and writing text, the class structure behind them, and how to close resources safely.

8.4.1 Reading a file with `Scanner`

`Scanner` can read not only the keyboard but also a file or a `String`. To change the source, a `File` object is passed to the constructor instead of `System.in`. The reading methods are the same. The following program reads a file line by line and prints it to the screen. The required imports are `java.io.File` , `java.io.FileNotFoundException` , and `java.util.Scanner` .

```
public class ScannerFile {
  public static void main(String[] args) {
    Scanner scanner = null;
    try {
      scanner = new Scanner(new File("sample.txt"));
      // Read the file and print it
      while (scanner.hasNextLine()) {
        String line = scanner.nextLine();
        System.out.println(line);
      }
    } catch (FileNotFoundException e) {
      System.err.println("File not found: " + e.getMessage());
    } finally {
      if (scanner != null)
        scanner.close();
    }
  }
}
```

Let `sample.txt` contain three lines: `Java`, `Input Output`, and `Streams`. The program prints them as they are:

```
Java
Input Output
Streams
```

Three points of the program deserve explanation:

- **A checked exception.** `new Scanner(File)` throws `FileNotFoundException` if the file does not exist. This is a checked exception: it is either caught or declared in the method header. The program catches it and writes a message to the error stream.
- **The loop.** `hasNextLine()` asks whether there is a line to read, and `nextLine()` takes the line. At the end of the file `hasNextLine()` returns `false` and the loop ends.
- `finally` **and the** `null` **check.** If the file cannot be opened, the variable `scanner` stays `null`. The `finally` block runs in every case; without the check, `close()` would be called through a `null` reference.

The last point is more important than it looks. If the line `if (scanner != null)` is removed and the program tries to open a nonexistent file, it first prints the "File not found" message and then crashes in `finally`:

```
File not found: missing.txt (The system cannot find the file specified)
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"java.util.Scanner.close()" because "<local1>" is null
        at NoNullCheck.main(NoNullCheck.java:12)
```

This pattern is both long and error-prone. The **try-with-resources** statement, introduced in Java 7, does the same job briefly and safely. `Scanner` implements the `AutoCloseable` interface, so it can be opened in the parentheses of `try` and is closed automatically when the block ends. Neither `finally` nor the `null` check is needed:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
public class TryWithResources {
  public static void main(String[] args) {
    try (Scanner scanner = new Scanner(new File("sample.txt"))) {
      while (scanner.hasNextLine()) {
        System.out.println(scanner.nextLine());
      }
    } catch (FileNotFoundException e) {
      System.err.println("File not found: " + e.getMessage());
    }
    Scanner s = new Scanner("3 4,5 apple\nsecond line");
    System.out.println(s.nextInt() + " | " + s.next() + " | " + s.next() + " | " +
s.nextLine() + " | " + s.nextLine());
  }
}
```

```
Java
Input Output
Streams
3 | 4,5 | apple |  | second line
```

The two lines at the end of the program show that `Scanner` can also read a `String`. `nextInt()` takes the 3, and the two `next()` calls take the words `4,5` and `apple`, each up to the next space. The first `nextLine()` that follows returns the rest of the line, which is the empty string; the second `nextLine()` returns the actual second line. This is exactly the `nextInt` – `nextLine` trap seen with the keyboard.

8.4.2 Code trace: try-with-resources and closing order

Definition 8.6

try-with-resources is the form of `try` that automatically closes the resources opened in the parentheses after the `try` keyword when the block ends. Only objects that implement the `AutoCloseable` interface can be opened in the parentheses. Resources are closed in the reverse of the order in which they were opened; closing happens whether the block ends normally or with an exception.

To see the closing order, we write a small resource class that prints a message when it is opened and when it is closed. `Res` implements the `AutoCloseable` interface; its only method is `close()`.

```
class Res implements AutoCloseable {
  String name;
  Res(String name) { this.name = name; System.out.println(name + " opened"); }
  public void close() { System.out.println(name + " closed"); }
}
public class CloseOrder {
  public static void main(String[] args) {
    try (Res a = new Res("A"); Res b = new Res("B")) {
      System.out.println("block running");
      throw new IllegalStateException("error!");
    } catch (IllegalStateException e) {
      System.out.println("caught: " + e.getMessage());
    } finally { System.out.println("finally"); }
  }
}
```

```
A opened
B opened
block running
B closed
A closed
caught: error!
finally
```

Table 8.8 gives the eight steps of the program.

Step	Line	Variables	New output
1	8: `try (Res a = new Res("A"); Res b = new Res("B")) {`	a=Res("A")	`A opened`
2	8: `try (Res a = new Res("A"); Res b = new Res("B")) {`	a=Res("A"), b=Res("B")	`B opened`

Step	Line	Variables	New output
3	9: `System.out.println("block running");`	a=Res("A"), b=Res("B")	`block running`
4	10: `throw new IllegalStateException("error!");`	a=Res("A"), b=Res("B")	—
5	4: `public void close() { System.out.println(name + " closed"); }`	this.name="B"	`B closed`
6	4: `public void close() { System.out.println(name + " closed"); }`	this.name="A"	`A closed`
7	11: `} catch (IllegalStateException e) {` 12: `System.out.println("caught: " + e.getMessage());`	e=IllegalStateException("error!")	`caught: error!`
8	13: `} finally { System.out.println("finally"); }`	—	`finally`

Table 8.8: The order in which two resources are opened and closed with try-with-resources. Both resources are closed before the exception in the block is caught.

In the first two steps the resources in the parentheses are created in order from left to right: first `a`, then `b`. In the third step the body of the block runs; both resources are open. In the fourth step the body throws an exception.

The fifth and sixth steps are the heart of this construct. Although an exception has been thrown, control does not pass to `catch` immediately. First the resources are closed, and they are closed in **reverse order**: `b`, opened last, closes first, then `a`. The reverse order has a reason. A resource opened later may depend on an earlier one; for example, a writer may wrap a file stream. Closing the outer one first lets it flush its buffer while the inner stream is still open.

In the seventh step `catch` catches the exception. At this point `a` and `b` are out of scope: variables declared in the parentheses are visible only inside the `try` block. A use such as `a.close()` inside `catch` or after the block does not compile. In the eighth step `finally` runs. The full order is: open A, open B, body, close B, close A, `catch`, `finally`.

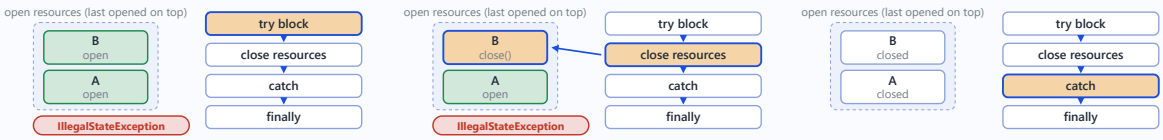


Figure 8.8: Three moments of the closing order. (a) The body throws an exception; both resources are open. (b) First `b`, opened last, closes. (c) After both resources are closed, `catch` catches the exception.

8.4.3 The `java.io` library

The `java.io` package offers a powerful and flexible toolkit for input and output. It is used in a wide range of areas, from file operations to network programming. The package provides flexibility along three axes:

- **Input and output:** it receives data from sources such as files, networks, and memory, and sends data to them.
- **Binary and text:** it supports both byte-based (binary) and character-based (text) operations.
- **Sequential and random:** it reaches the data in a file both in order and by jumping to a chosen position.

The elements of the package fall into three groups (Table 8.9).

Kind of element	Examples	Purpose
Classes	`File`	operations on files and directories (paths)
	`InputStream`, `OutputStream`	basic byte input and output
	`Reader`, `Writer`	text (character) input and output
Interfaces	`Closeable`, `AutoCloseable`	releasing a resource: `close()`
	`Flushable`	emptying the buffer: `flush()`
Exceptions	`IOException`	the superclass of I/O errors
	`FileNotFoundException`, `EOFException`	file not found / unexpected end of file

Table 8.9: The basic elements of the `java.io` package.

8.4.4 Byte streams and character streams

The streams in `java.io` fall into two large families. The distinction is made by the unit the stream carries.

Definition 8.7

Byte streams process data as 8-bit bytes; their superclasses are `InputStream` and `OutputStream`. **Character streams** process data as 16-bit Unicode `char` values; their superclasses are `Reader` and `Writer`. Character streams convert bytes into characters, and characters into bytes, with a character encoding.

Byte streams work with binary data such as images, audio, video, and `.class` files. `read()` returns a byte between 0 and 255 and does not care what the byte means. Character streams are designed for text-based data. Because they decode the encoding, they handle different languages and alphabets correctly. Table 8.10 compares the two families.

	Byte stream	Character stream
Unit read	`byte` (8 bits)	`char` (16 bits)
Superclasses	`InputStream`, `OutputStream`	`Reader`, `Writer`
File classes	`FileInputStream`, `FileOutputStream`	`FileReader`, `FileWriter`
Buffered	`BufferedInputStream`, `BufferedOutputStream`	`BufferedReader`, `BufferedWriter`

	Byte stream	Character stream
Use	any kind of file, copying	text files only

Table 8.10: Comparison of byte streams and character streams.

Reading text with a byte stream is possible but hard. The following experiment reads the word `Işık` byte by byte and converts each byte to a character with `(char)`:

```
IÅÄ±k (6 char)
```

A four-letter word comes out as six "characters". `ş` and `ı` are two bytes each in UTF-8, and a byte stream delivers these bytes separately. Converting every byte to a `char` on its own produces wrong letters. A character stream, by contrast, decodes the two bytes together and returns the right letter.

There are bridge classes between the two families. `InputStreamReader` turns a byte stream into a character stream, and `OutputStreamWriter` turns characters into a byte stream. The encoding is applied in these bridges. The `writeText` method that wrote a number in UTF-16 used this bridge.

8.4.5 Class hierarchies

Each of the two families consists of classes derived from an abstract superclass. Figure 8.9 shows the byte streams and Figure 8.10 the character streams.

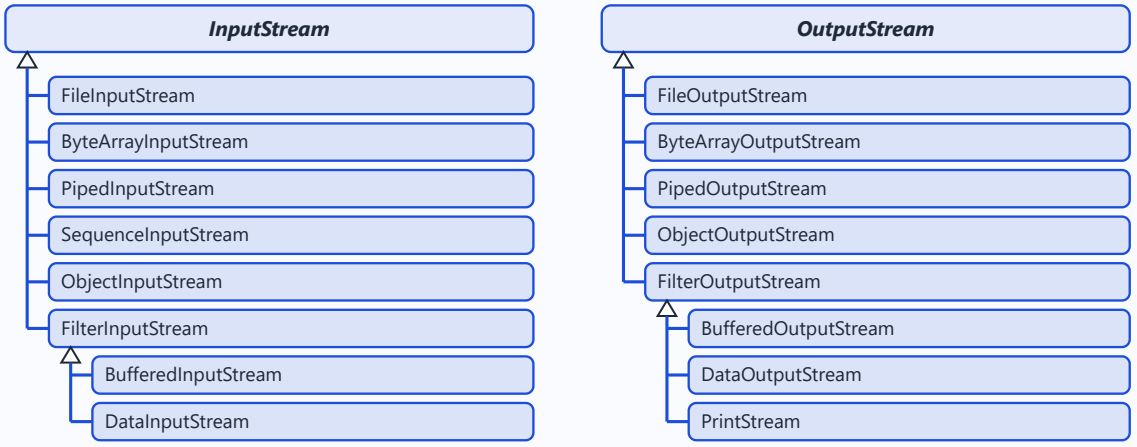


Figure 8.9: Byte stream classes. Italic boxes are abstract classes; arrows with hollow triangles show inheritance. `System.out` is a `PrintStream`.

`InputStream` and `OutputStream` at the top of the hierarchy are abstract classes, so no objects are created from them directly. The subclasses take on two kinds of work. Some connect to a **data source**: `FileInputStream` connects to a file, `ByteArrayInputStream` to an array in memory, and `PipedInputStream` to a pipe. The others derive from `FilterInputStream` and `FilterOutputStream`.

These do not connect to a source on their own; they wrap another stream and add a new capability to it. `BufferedInputStream` adds a buffer, and `DataInputStream` adds the ability to read primitive types.

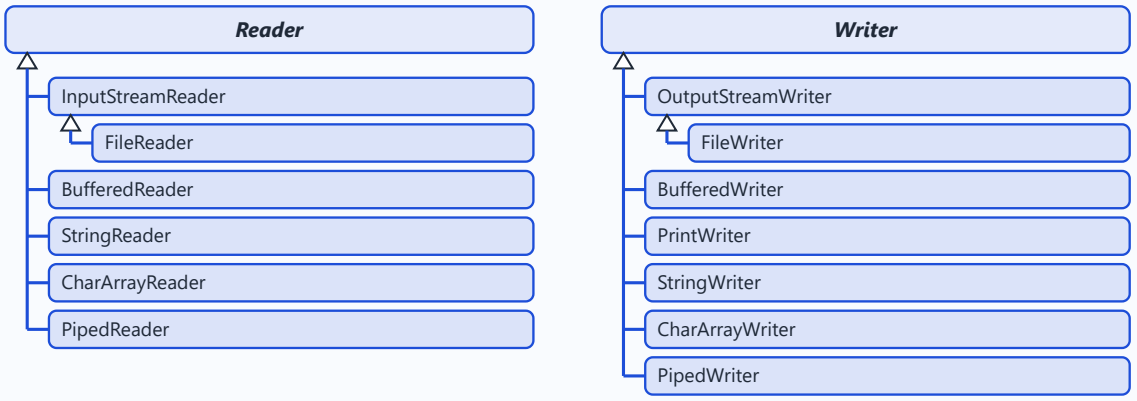


Figure 8.10: Character stream classes. A `FileReader` is an `InputStreamReader`; `FileWriter` likewise derives from `OutputStreamWriter`.

Character streams follow the same pattern. The relationship that stands out here is the position of `FileReader`. A `FileReader` is really an `InputStreamReader`: inside, it opens a `FileInputStream` and converts the bytes coming from it into characters with the chosen encoding. `FileWriter` likewise derives from `OutputStreamWriter`. So underneath every character stream there is, in the end, a byte stream. A disk only knows bytes.

8.4.6 Wrapper streams

Every class in Java's I/O library does one job well. `FileReader` reads characters from a file but does not recognize lines. `BufferedReader` reads lines but cannot connect to a file. The capabilities you need are combined by nesting streams inside one another.

Definition 8.8

A **wrapper stream** (also called a *decorator*) is a stream that takes another stream in its constructor and adds a new capability to it, such as buffering, line reading, or reading of primitive types. The wrapper's methods do part of the work themselves and pass the rest on to the inner stream. This arrangement is known as the Decorator design pattern.

```
BufferedReader reader = new BufferedReader(new FileReader("sample.txt"));
PrintWriter writer = new PrintWriter(new BufferedWriter(new FileWriter("file2.txt")));
```

The first line has two layers: `FileReader` opens the file and converts bytes to characters; `BufferedReader` wraps it and adds a buffer and `readLine()`. The second line has three layers: `FileWriter` converts characters to bytes and writes them to the file, `BufferedWriter` adds a buffer, and `PrintWriter` adds convenient writing methods such as `print`, `println`, and `printf` (Figure 8.11).

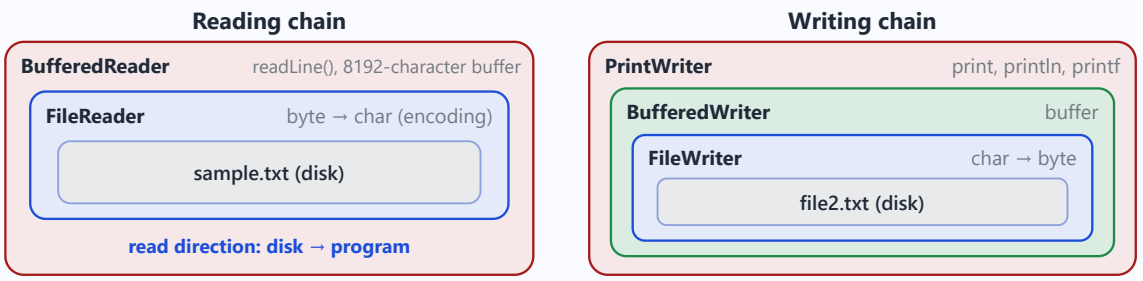


Figure 8.11: Reading and writing chains. Data passes layer by layer from the outer stream to the inner one, or from the inner one to the outer one.

You can think of a garden hose: first a filter and then a spray nozzle are attached to the end of the hose. The water, which is the data, passes through all of them in turn. Each part does its own job, and the parts can be combined in any order you like. To add a capability, you do not need to write a new class; you combine the existing wrappers.

Wrapping has two rules. The first: **closing the outermost stream closes all the inner ones**. `writer.close()` first flushes its own buffer, then the buffer of the `BufferedWriter`, and finally closes the `FileWriter` and the file. The inner streams do not have to be closed separately. The second: **a wrapper interprets the data but does not validate it**. If a text file is wrapped in a `DataInputStream` and `readInt()` is called, the first four bytes, the codes of the letters `J`, `a`, `v`, `a`, are read as the integer 1,247,901,281. The stream raises no error; the wrong wrapper produces a wrong but valid-looking result.

8.4.7 Code trace: `BufferedReader`

The most common way to read a text file line by line is the `readLine()` method of `BufferedReader`. The method returns the next line without its line terminator; when the file ends, it returns `null`.

```
import java.io.*;

public class ReadLines {
  public static void main(String[] args) {
    File file = new File("sample.txt");
    try (BufferedReader reader =
        new BufferedReader(new FileReader(file))) {
      String line;
      while ((line = reader.readLine()) != null) {
        System.out.println(line);
      }
    } catch (IOException e) {
      System.err.println("Read error: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

The expression `(line = reader.readLine()) != null` in the loop condition is a common pattern in Java I/O code. First the assignment in the parentheses is performed: the line read is stored in the variable `line`. The value of an assignment expression is the assigned value itself. This value is compared with `null`. So reading and the end-of-file check happen on one line. The outer parentheses are required. Without them `!=` would run first and a `boolean` value would be assigned to a `String` variable; the code would not compile. When the file contains the lines `Java`, `Input Output`, and `Streams`, the trace is as in Table 8.11.

Step	Line	Variables	New output
1	5: `File file = new File("sample.txt");`	file="sample.txt"	—
2	6: `try (BufferedReader reader =` 7: `new BufferedReader(new FileReader(file))) {`	reader=BufferedReader	—
3	9: `while ((line = reader.readLine()) != null) {`	line="Java"	—
4	10: `System.out.println(line);`	line="Java"	`Java`
5	9: `while ((line = reader.readLine()) != null) {`	line="Input Output"	—
6	10: `System.out.println(line);`	line="Input Output"	`Input Output`
7	9: `while ((line = reader.readLine()) != null) {`	line="Streams"	—
8	10: `System.out.println(line);`	line="Streams"	`Streams`
9	9: `while ((line = reader.readLine()) != null) {`	line=null	—
10	12: `} catch (IOException e) {`	—	—

Table 8.11: Trace of the `ReadLines` program. The loop condition (line 9) reads one line in each round; at the end of the file it gets `null`.

In the first step only a `File` object is created; the file has not been opened yet. In the second step `FileReader` opens the file and `BufferedReader` wraps it. The read position is at the start of the file. In the third step the first call to `readLine()` returns `"Java"`. By this time `BufferedReader` has read a large chunk of the file in one go and put it in its buffer; for a small file the whole file fits into the buffer. The following lines are served from the buffer without going to disk.

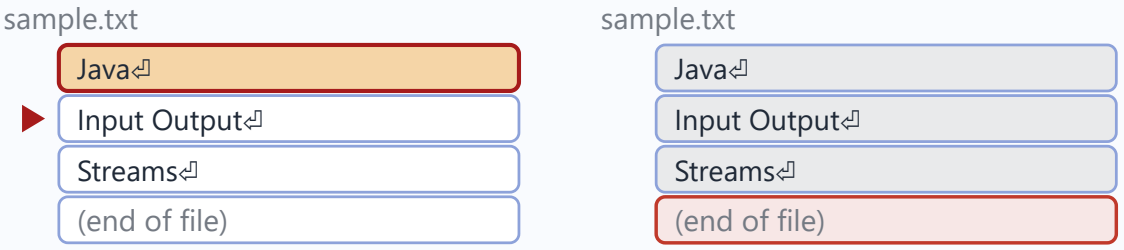


Figure 8.12: Two moments from the `readLine()` trace. (a) The first line is read and its line terminator is discarded. (b) At the end of the file `readLine()` returns `null` and the loop ends.

In the ninth step the file ends and `readLine()` returns `null`. The loop terminates. In the tenth step the `try` block ends and the reader closes automatically. Since there is no error, `catch` does not run. Because every line read is written once, the running time of the program is proportional to the size of the file. In memory, only one line and the buffer are held at a time; memory use is constant however large the file is. This property makes it possible to process line by line even log files that are gigabytes in size.

Common mistake

Calling `readLine()` twice inside the loop: `while (reader.readLine() != null) System.out.println(reader.readLine());`. The call in the condition reads a line and throws it away, and the call in the body prints the next one. Half of the lines are lost, and for a file with an odd number of lines `null` is printed at the end. Each line is read once and kept in a variable.

8.4.8 `PrintWriter`

`PrintWriter` is the most convenient way to write to text files. It has all the methods familiar from `System.out`: `print`, `println`, and `printf`. It converts numbers, characters, `boolean` values, and other types to text itself and writes them.

```
public class PrintWriterDemo {
  public static void main(String[] args) {
    PrintWriter writer = null;
    try {
      writer = new PrintWriter("newFile.txt");
      writer.println("Hello, world!");
      writer.println("This is a sample file.");
      writer.printf(Locale.ROOT, "Pi is about %.2f%n", Math.PI);
      writer.println(42);
    } catch (IOException e) {
      System.err.println("File error: " + e.getMessage());
      e.printStackTrace();
    } finally {
      if (writer != null)
        writer.close();
    }
  }
}
```

After the program has run, the contents of `newFile.txt` are:

```
Hello, world!
This is a sample file.
Pi is about 3.14
42
```

`new PrintWriter("newFile.txt")` creates the file if it does not exist. If the file exists, it **erases its contents** and writes from the beginning. If the file cannot be created, the constructor throws `FileNotFoundException`; the program catches it as an `IOException`. In `printf`, `%.2f` writes the number with two decimal places and `%n` writes the operating system's line terminator. The first argument, `Locale.ROOT` (`java.util.Locale`), makes the formatting independent of the locale.

Here is why `Locale.ROOT` is passed. If `printf` is called without a locale, it chooses the decimal separator according to the computer's **locale**. On a Turkish system it writes `3,14`, on an English system `3.14`. The same program produces different files on two machines. If a program is going to read the file later, this difference causes trouble: `Double.parseDouble("3,14")` fails. With `Locale.ROOT` the output is `3.14` on every machine. `println(3.5)`, on the other hand, is already independent of the locale and always writes a period. The following experiment was run in the Turkish locale:

```
import java.io.*;
import java.util.Locale;
public class LocaleDemo {
  public static void main(String[] args) throws Exception {
    try (PrintWriter p = new PrintWriter("loc.txt")) {
      p.printf("Pi is about %.2f%n", Math.PI);
      p.printf(Locale.ROOT, "Pi is about %.2f%n", Math.PI);
      p.println(3.5);
      p.println('A');
      p.println(true);
    }
    Show.main(new String[] {"loc.txt"});
  }
}
```

```
Pi is about 3,14
Pi is about 3.14
3.5
A
true
[50 bytes]
```

`PrintWriter` has one more important property: its writing methods do not throw exceptions. If an I/O error occurs while writing, `PrintWriter` records it in an internal flag and silently continues. The error becomes visible only by calling `checkError()`. The trace in the buffering section shows this behavior. Finally, do not forget `close()`: `PrintWriter` contains a buffer, and if it is not closed the last things written may never reach the disk.

8.4.9 `FileWriter`: writing and appending

`FileWriter` is the basic class for writing characters to files; it connects the writing function of the `Writer` class to a file. It has only the `write(String)`, `write(char[])`, and `write(int c)` methods. Its text-formatting ability is limited: the line terminator (`\n`) is written separately, and numbers are first converted to strings. It is a lower-level and more flexible approach, which is why it is mostly wrapped in a `BufferedWriter` or a `PrintWriter`.

`write(int c)` is often confused. This method writes not the number but the **character** whose code is that number. `w.write(65)` writes `A` to the file; to write `65` you need `w.write(String.valueOf(65))`. The `A65` in the last output line of the program below is the result of these two calls.

The most useful capability of `FileWriter` is the **append** mode. `new FileWriter(name, true)` adds to the end of the file without erasing it. If `false` is passed, or the second parameter is omitted, the file is reset.

```
public class FileWriterDemo {
  public static void main(String[] args) {
    try (FileWriter writer = new FileWriter("newFile.txt")) {
      writer.write("Hello, world!\n");
      writer.write("This is a FileWriter example.\n");
      System.out.println("File created and data written.");
    } catch (IOException e) {
      System.err.println("File error: " + e.getMessage());
      e.printStackTrace();
    }
    // true: append to the end of the file
    try (FileWriter app = new FileWriter("newFile.txt", true)) {
      app.write("Appended line.\n");
    } catch (IOException e) {
      System.err.println("File error: " + e.getMessage());
    }
  }
}
```

```
File created and data written.
```

The first `try` block creates the file and writes two lines. The second block opens the file with `true`, so it keeps the two existing lines and appends the third line at the end. When the program ends, `newFile.txt` is:

```
Hello, world!
This is a FileWriter example.
Appended line.
```

```
Hello, world!
This is a FileWriter example.
Appended line.
[59 bytes]
A65
```

The file is 59 bytes: the three lines have 56 characters, and the three line terminators add 3. All the characters are ASCII, so each takes one byte; letters such as `ü` would take two bytes each in UTF-8. Append mode is indispensable for log files: each time the program runs, the new records are added after the old ones. If `true` had been forgotten in the second `try` block, the first two lines would be erased and only “Appended line.” would remain in the file. Table 8.12 compares the three writer classes.

	`FileWriter`	`BufferedWriter`	`PrintWriter`
Writing methods	`write`	`write`, `newLine()`	`print`, `println`, `printf`
Writing a number	converted to a `String` first	converted to a `String` first	directly: `println(42)`
Buffer	small internal buffer	8192-character buffer	depends on the stream it wraps
Error reporting	`IOException`	`IOException`	`checkError()`

Table 8.12: `FileWriter`, `BufferedWriter`, and `PrintWriter`.

The choice depends on the job. If a formatted report or table is to be written, use `PrintWriter`; if many short writes are to be made, put a `BufferedWriter` in between. If errors should be reported at once as exceptions, `BufferedWriter` is preferable; with `PrintWriter` an error shows up only when `checkError()` is asked.

8.4.10 Specifying the character encoding

Character streams that are not given an encoding use the default encoding. In Java 18 and later this is UTF-8. In older versions it was the operating system's setting; on Turkish Windows, Windows-1254. The same file could be read differently on two machines. In portable code the encoding is given explicitly. Since Java 11, `FileReader` and `FileWriter` have constructors that take an encoding. In Java 10 and earlier the same job was done with the bridge classes `InputStreamReader` and `OutputStreamWriter`.

```
public class EncodingRead {
  public static void main(String[] args) throws IOException {
    File file = new File("turkish.txt");
    try (BufferedWriter writer = new BufferedWriter(
        new FileWriter(file, StandardCharsets.UTF_8))) {    // Java 11+
      writer.write("Işık, şimşek, çiğ, öğrenci, Ünye");
    }
    try (BufferedReader right = new BufferedReader(
        new FileReader(file, StandardCharsets.UTF_8))) {
      System.out.println(right.readLine());
    }
    try (BufferedReader wrong = new BufferedReader(
        new FileReader(file, StandardCharsets.ISO_8859_1))) {
      System.out.println(wrong.readLine());
    }
  }
}
```

```
Işık, şimşek, çiğ, öğrenci, Ünye
IÅÄ±k, ÅimÅek, Ã§iÄ, Ã¶Ärenci, Ãnye
```

When the file is written in UTF-8 and read in UTF-8, the text comes back correctly. When the same bytes are read as ISO-8859-1, every Turkish letter turns into two separate characters. For example, the UTF-8 bytes of `ş` are `C5 9F`. In Latin-1, `C5` is the letter `Å`, while `9F` is a non-printable control character that does not show on the screen. So `şimşek` appears as `ÅimÅek`. This kind of garbling is called *mojibake*. An encoding error does not raise an exception; the program runs, but the data comes back corrupted.

Information is also lost when a character with no equivalent is written. If `ş` is given to a stream that writes ASCII, a question mark (`?`, `3F`) is written, because ASCII has no equivalent for this letter. In UTF-8 the same letter would have been stored as `C5 9F`. This loss cannot be undone: the file now contains only a question mark.

Common mistake

Writing a file in one encoding and reading it in another. If a file produced by an old Windows program in Windows-1254 is read in Java 18 or later without naming an encoding, the Turkish letters turn into the `�` sign. If the encoding of the file is known, give it explicitly when reading: `new FileReader(file, Charset.forName("windows-1254"))`.

8.5 Buffering

The basic methods of streams work with small pieces. `read()` reads a byte or a byte array, and `write()` writes a byte or a byte array. If the program goes to the disk separately for every byte, the application slows down badly. This section explains the cause of the problem and its solution.

8.5.1 Why buffer?

A disk is thousands of times slower than memory. Moreover, a disk access is not just transfer time. The program makes a **system call**, the operating system runs the device driver, and the result returns to the program. This fixed cost is roughly the same whether one byte or eight thousand bytes are read. Reading a thousand bytes with a thousand separate calls is therefore far more expensive than reading them with one call.

Definition 8.9

A **buffer** is an area in which data is temporarily collected in memory. **Buffering** is the technique of reducing the number of physical disk accesses by doing reads and writes in bulk through this area. When reading, the buffer is filled with a large piece at a time, and the program's small reads are served from the buffer. When writing, small writes accumulate in the buffer, and when the buffer is full it is sent to the disk in one go.

An everyday analogy explains the difference. To buy ten items at the market, you do not make a separate trip for each item; you put them all in one basket and carry them at once. The trip itself, the fixed cost, is paid once. Figure 8.13 compares unbuffered and buffered writing.

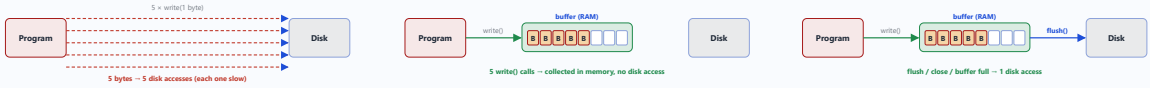


Figure 8.13: Unbuffered and buffered writing. (a) In an unbuffered stream, every `write()` goes straight to the disk. (b) In a buffered stream, `write()` puts the bytes into the buffer in memory. (c) When the buffer is full, or when `flush()` or `close()` is called, the whole buffer is written to disk at once.

8.5.2 Buffered stream classes and `flush`

In Java, buffering is added with wrapper streams. There are four buffered classes (Table 8.13).

Class	Stream it wraps	What it does
`BufferedInputStream`	`InputStream`	Speeds up reading: when the buffer runs empty it reads 8192 bytes from the disk at once.
`BufferedReader`	`Reader`	Does the same for characters; also adds `readLine()`.
`BufferedOutputStream`	`OutputStream`	Speeds up writing: collects the data of `write()` in the buffer.
`BufferedWriter`	`Writer`	Does the same for characters; also adds `newLine()`.

Table 8.13: Buffered stream classes.

A write buffer has a side effect: when `write()` returns, the data may not yet be on the disk. There are three moments when data goes to the disk. When the buffer fills up, it is emptied automatically. When `flush()` is called, the buffer is emptied to the disk immediately. When `close()` is called, `flush()` runs first and then the stream is closed. The following program shows this behavior by measuring the file size:

```
public class FlushDemo {
  public static void main(String[] args) throws IOException {
    File log = new File("log.txt");
    BufferedWriter bw = new BufferedWriter(new FileWriter(log));
    bw.write("Record 1");
    System.out.println("before: " + log.length());
    bw.flush();
    System.out.println("after: " + log.length());
    bw.close();
  }
}
```

```
before: 0
after: 8
```

After `bw.write("Record 1")` returns, the file is still 0 bytes; the eight characters wait in the buffer of the `BufferedWriter`. After `flush()` the file is 8 bytes, one byte per character, because all of them are ASCII.

Common mistake

Not closing a buffered writer. If you write `bw.write("data");` and never close the stream, the data stays in the buffer. The buffer is not flushed when the program ends either: when the JVM exits, it does not write the buffers of open streams to disk. The file looks empty. Call `close()` or `flush()`; the safest way is try-with-resources.

This mistake can really be measured. The following two programs were run one after the other. The first writes and ends without closing; the second looks at the size of the file:

```
note.txt size: 0
```

```
program finished, note.txt size: 0
```

Even after the program has ended, the file is 0 bytes. The five characters never reached the disk.

8.5.3 Code trace: the buffer and `flush`

The following code fragment traces step by step the buffer beneath a `PrintWriter`. After every write, the size of the file is measured. The last two lines show what happens when you write to a closed stream.

```
File f = new File("log.txt");
PrintWriter pw = new PrintWriter(new FileWriter(f));
System.out.println("0: " + f.length());
pw.print("entry1;");
System.out.println("1: " + f.length());
pw.print("entry2;");
System.out.println("2: " + f.length());
pw.flush();
System.out.println("3: " + f.length());
pw.print("entry3;");
pw.close();
System.out.println("4: " + f.length());
pw.print("entry4;");   // closed: silently ignored
System.out.println("error? " + pw.checkError());
```

```
0: 0
1: 0
2: 0
3: 14
4: 21
error? true
```

Step	Line	Variables	New output
1	1: `File f = new File("log.txt");` 2: `PrintWriter pw = new PrintWriter(new FileWriter(f));`	f=log.txt, pw=PrintWriter	—
2	3: `System.out.println("0: " + f.length());`	f.length()=0	`0: 0`
3	4: `pw.print("entry1;");`	buffer="entry1;"	—
4	5: `System.out.println("1: " + f.length());`	f.length()=0	`1: 0`
5	6: `pw.print("entry2;");` 7: `System.out.println("2: " + f.length());`	buffer="entry1;entry2;"	`2: 0`
6	8: `pw.flush();`	buffer=""	—
7	9: `System.out.println("3: " + f.length());`	f.length()=14	`3: 14`
8	10: `pw.print("entry3;");`	buffer="entry3;"	—
9	11: `pw.close();`	pw=closed	—
10	12: `System.out.println("4: " + f.length());`	f.length()=21	`4: 21`

11	13: `pw.print("entry4;"); // closed: silently ignored`
	14: `System.out.println("error? " + pw.checkError());`
		pw.checkError()=true	`error? true`

Table 8.14: Trace of the buffer and `flush`. The "Variables" column gives the contents of the buffer and the measured file size.

In the first step of Table 8.14, `FileWriter` creates the file; the file is empty. In this chain the buffer lives inside `FileWriter`. `FileWriter` uses an encoder that converts characters into bytes, and this encoder has a byte buffer. `PrintWriter` adds no buffer of its own; it passes what is written on to this

buffer.

In the third and fifth steps, the `print` calls put seven characters each into the buffer but do not touch the disk. The file stays at 0 bytes. In the sixth step `flush()` empties the buffer to disk and 14 bytes are written. In the eighth step the third entry goes into the buffer. The `close()` in the ninth step first does a `flush()` and then closes the file. In the tenth step the file is 21 bytes. Had `close()` been forgotten, the third entry could have been lost.

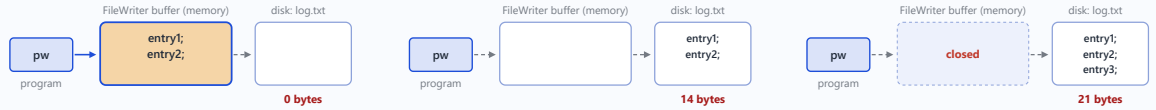


Figure 8.14: Three moments of the buffer. (a) Two entries wait in the buffer; the file is 0 bytes. (b) After `flush()` the file is 14 bytes. (c) After `close()` the file is 21 bytes.

The eleventh step shows that `PrintWriter` does not throw exceptions. Calling `print` on a closed stream would normally require an `IOException`. `PrintWriter` does not fail loudly; the write silently fails and an internal flag is set. `checkError()` returns `true`. This behavior suits `System.out`, because wrapping every `println` in a `try` would be tedious. When writing to an important file, however, `checkError()` should be checked at the end, or a writer that throws exceptions should be chosen.

8.5.4 Buffered reading

On the reading side, buffering is used in two ways: byte by byte and line by line. Suppose `file.txt` contains two lines: `Java I/O` and `Buffer`.

```
try {
  FileInputStream fis = new FileInputStream("file.txt");
  BufferedInputStream bis = new BufferedInputStream(fis);
  int b;
  while ((b = bis.read()) != -1) {
    System.out.print((char) b);
  }
  bis.close();     // also closes fis
} catch (IOException e) {
  System.err.println("File error: " + e.getMessage());
}
```

Output: reading byte by byte

```
Java I/O
Buffer
```

This loop calls `bis.read()` once for every byte. But the disk is not read one byte at a time. On the first call the buffer is empty; `BufferedInputStream` reads up to 8192 bytes at once from the underlying `FileInputStream`. Later calls are served from the buffer without going to the disk. The line terminator is a byte too, and when printed with `(char)` it ends the line. That is why the output has two lines. This method gives correct results only for ASCII text; Turkish letters are broken because they take two bytes each.

```
try {
  FileReader fr = new FileReader("file.txt");
  BufferedReader br = new BufferedReader(fr);
  String line;
  while ((line = br.readLine()) != null) {
    System.out.println(line);
  }
  br.close();     // also closes fr
} catch (IOException e) {
  System.err.println("File error: " + e.getMessage());
}
```

The second fragment reads the same file line by line with `BufferedReader` and prints the same two lines. Because the character stream decodes the bytes, Turkish letters come through correctly too. In both fragments `close()` is applied only to the outermost stream: `bis.close()` closes the inner `fis`, and `br.close()` closes the inner `fr`.

These two fragments share a weakness. If an exception occurs during reading, the `close()` line is never reached and the file stays open. For this reason most of the programs in the rest of the chapter use try-with-resources.

8.5.5 How much does buffering help?

The following program creates a 2 MB file and reads it twice byte by byte: first unbuffered, then buffered. The time is measured in milliseconds with `System.nanoTime()`.

```
public class BufferTiming {
  static long time(InputStream in) throws IOException {
    long start = System.nanoTime();
    while (in.read() != -1) { }          // 1 byte at a time
    in.close();
    return (System.nanoTime() - start) / 1_000_000;
  }
  public static void main(String[] args) throws IOException {
    File f = new File("big.dat");
    try (FileOutputStream fos = new FileOutputStream(f)) {
      fos.write(new byte[2 * 1024 * 1024]);   // 2 MB
    }
    long t1 = time(new FileInputStream(f));
    long t2 = time(new BufferedInputStream(new FileInputStream(f)));
    System.out.println("Unbuffered: " + t1 + " ms");
    System.out.println("Buffered  : " + t2 + " ms");
    f.delete();
  }
}
```

The output of the sample run in the slides is:

```
Unbuffered: 3151 ms
Buffered  : 94 ms
```

While these notes were being prepared, the same program was run three more times. The times were 3194/58, 3241/62, and 3250/63 milliseconds. The times vary with the machine, the disk, and the operating system's cache; the ratio stays similar. On this machine buffered reading is more than fifty times faster.

The reason for the difference is the number of calls. The file is 2 MB, that is, $2 \cdot 1024 \cdot 1024 = 2{,}097{,}152$ bytes. In the unbuffered stream every `read()` is a system call: 2,097,152 calls, plus one more that reports the end of the file. In the buffered stream there is one system call per 8192 bytes: 2,097,152 / 8192 = 256 calls that carry data and one call for the end of the file. The number of the program's own `read()` calls is the same in both cases; but in the buffered case these are ordinary method calls that read from an array in memory. The number of system calls drops by a factor of about 8000. The time drops only by a factor of about fifty, because a method call is still made for every byte. Reading block by block removes this cost as well.

8.6 Other byte streams

The `InputStream` and `OutputStream` family has classes besides the file streams. They use an array in memory, another thread, or several streams as the source of data. Since they all carry the same `read()` and `write()` methods, they can be used wherever a file stream is expected.

8.6.1 `FileInputStream` and reading in chunks

`FileInputStream` derives from `InputStream` and reads bytes directly from a file. It is a low-level class, suited to non-text data such as images and sound. It has two read methods:

- `read()` reads a single byte and returns an `int` between 0 and 255. At the end of the file it returns −1.
- `read(byte[] b)` reads at most `b.length` bytes into the array and returns **the number of bytes it read**. At the end of the file it returns −1. The returned number may be smaller than the length of the array.

```
public class ChunkRead {
  public static void main(String[] args) throws IOException {
    try (FileInputStream fis = new FileInputStream("file.txt")) {
      byte[] buf = new byte[4];
      int n;
      while ((n = fis.read(buf)) != -1) {
        System.out.println(n + ": " + new String(buf, 0, n));
      }
    }
  }
}
```

Suppose `file.txt` contains `Hello, Java!` : 12 bytes. Because the array holds 4 bytes, the file is read in three chunks:

```
4: Hell
4: o, J
4: ava!
```

The second line, `o, J`, contains a comma and a space. `new String(buf, 0, n)` converts only the part of the array that was read, that is, `n` bytes starting at 0. In this example the file is an exact multiple of 4, so every chunk is full. Had the file been 13 bytes, the fourth read would have returned 1, and the last three cells of the array would still hold bytes left over from the previous chunk. Had `new String(buf)` been written, these stale bytes would have become part of the text.

Using `FileInputStream` for text is difficult. Multibyte characters such as Turkish letters can be split at a chunk boundary. If the word `Gündüz` is read in 2-byte chunks and each chunk is decoded separately as UTF-8, the two bytes of `ü` land in two different chunks:

```
[G�][�n][d�][�z]
```

Each half-finished byte is an invalid UTF-8 sequence and turns into the `�` sign. For text files, character-based classes such as `FileReader` are more suitable; they keep a half character at a chunk boundary for the next read.

8.6.2 In-memory streams

`ByteArrayInputStream` also derives from `InputStream`, but it reads not from a file but from a byte array in memory. It is fast because it touches no physical resource. It can only be read; to write to memory, its counterpart `ByteArrayOutputStream` is used. `ByteArrayOutputStream` collects the bytes written in a growing array; `toByteArray()` returns this array and `toString()` its text equivalent.

```
public class ByteArrayRead {
  public static void main(String[] args) throws IOException {
    byte[] arr = {74, 97, 118, 97};           // "Java"
    ByteArrayInputStream bais = new ByteArrayInputStream(arr);
    int b;
    while ((b = bais.read()) != -1) {
      System.out.print(b + "=" + (char) b + " ");
    }
    System.out.println();
    // Writing to memory: ByteArrayOutputStream
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    baos.write(arr);
    baos.write('!');
    byte[] result = baos.toByteArray();
    System.out.println(result.length + " bytes: " + baos);
  }
}
```

```
74=J 97=a 118=v 97=a
5 bytes: Java!
```

The first loop reads the four bytes of the array. It prints each byte as a number and as a character: 74 = `J`, 97 = `a`, 118 = `v`. In the second part, four bytes and the character `'!'` are written to memory. The result is an array of 5 bytes.

These classes are useful in two situations. The first is testing a method that expects a stream without using files: a test can supply the desired input through a `ByteArrayInputStream` and collect the method's output in a `ByteArrayOutputStream`. The second is when the whole data is prepared in memory and then sent at once, for example over a network. Because all the data sits in memory, memory consumption becomes a problem with very large data.

8.6.3 A deprecated class

`java.io.StringBufferInputStream` is an old class written to read bytes from a string. It has been **deprecated** since Java 1.1. The class still compiles, but the compiler issues a warning and its use is discouraged. Its problem is that it does not convert characters to bytes correctly: it takes only the low 8 bits of each character.

```
public class OldStream {
  @SuppressWarnings("deprecation")
  public static void main(String[] args) throws IOException {
    InputStream old = new StringBufferInputStream("$€");
    int b;
    while ((b = old.read()) != -1) {
      System.out.print(b + " ");
    }
    System.out.println();
    byte[] right = "$€".getBytes("UTF-8");
    InputStream fresh = new ByteArrayInputStream(right);
    while ((b = fresh.read()) != -1) {
      System.out.print(b + " ");
    }
    System.out.println();
  }
}
```

```
95 172
197 159 226 130 172
```

The Unicode code of `$` is 351, hexadecimal `015F`. Its low 8 bits are `5F` = 95, which is the character `_`. The code of `€` is `20AC` and its low 8 bits are `AC` = 172. The two characters turn into two meaningless bytes and the real information is lost. The correct bytes are the UTF-8 bytes: 197 159 for `$` and 226 130 172 for `€`. These bytes are obtained with `"$€".getBytes("UTF-8")` and read with a `ByteArrayInputStream`. If the string has to be read as characters, `StringReader` is used. The `@SuppressWarnings("deprecation")` annotation silences the compiler warning in this example, which uses the old class on purpose.

8.6.4 Pipes

`PipedInputStream` and `PipedOutputStream` provide safe communication of data between two **threads**. A thread is an independently progressing flow of execution inside the same program. One thread writes data with `PipedOutputStream`, and the other reads it with `PipedInputStream`. The two ends are connected with `new PipedOutputStream(pis)` (Figure 8.15).

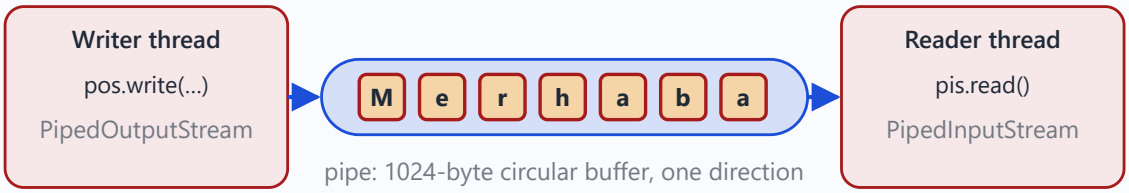


Figure 8.15: A pipe. The writer thread writes to the `PipedOutputStream`; the bytes wait in a circular buffer of 1024 bytes; the reader thread reads them from the `PipedInputStream`.

A pipe has four properties:

- Reading and writing are **synchronized**: if the pipe is empty the reader waits, and if it is full the writer waits.
- Data flows in **one direction**. Two-way communication needs two pipes.
- The buffer of a pipe is 1024 bytes by default.
- Using both ends **in the same thread** can cause a deadlock. When the buffer is full, the writer waits for the reader; but the reader is the same thread and never gets to run.

The following program creates a reader thread and a writer thread. The bodies of the threads are written as lambda expressions.

```
PipedInputStream pis = new PipedInputStream();
PipedOutputStream pos = new PipedOutputStream(pis);
```

```
// Reader thread
Thread reader = new Thread(() -> {
  try {
    int b;
    while ((b = pis.read()) != -1)
      System.out.print((char) b);
    System.out.println();
    pis.close();
  } catch (IOException e) {
    e.printStackTrace();
  }
});
```

```
// Writer thread
Thread writer = new Thread(() -> {
  try {
    pos.write("Hello, pipe!".getBytes());
    pos.close();
  } catch (IOException e) {
    e.printStackTrace();
  }
});
reader.start();
writer.start();
writer.join();
reader.join();
```

```
Hello, pipe!
```

`start()` starts a thread; `join()` waits for it to finish. The two threads run at the same time, and which of them makes progress first can change from run to run. If the reader starts first, it waits in `pis.read()` because the pipe is empty. The writer writes the 12 bytes into the pipe and calls `pos.close()`. The reader's waiting `read()` call picks up the bytes in order. When the pipe has been emptied and the writer end is closed, `read()` returns −1 and the loop ends. The output is the same in every run, because a pipe preserves the order of the bytes. The program was run twice for these notes and printed the same line both times.

To see the deadlock warning in action, the buffer of the pipe is reduced to 4 bytes and the writer tries to write 6 bytes. Nobody is reading:

```
writer still running after 1.5 s? true, bytes in pipe: 4
```

The writer writes the first 4 bytes and then waits because the pipe is full. After 1.5 seconds it is still waiting. Had the reader been the same thread, the program would have stopped forever. This experiment makes the writer a background thread so that the program can exit.

8.6.5 `SequenceInputStream`

`SequenceInputStream` joins several `InputStream` objects end to end into one sequential stream (Figure 8.16). When the first stream ends and returns −1, reading automatically moves on to the second. In this way data from different sources is processed as if it were a single stream.

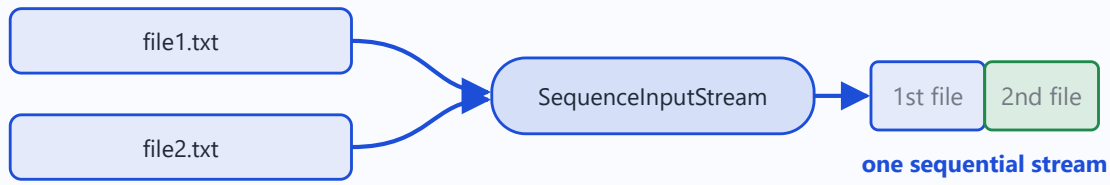


Figure 8.16: `SequenceInputStream` joins two file streams end to end. The reading program sees a single sequential stream.

```
try {
  FileInputStream is1 = new FileInputStream("file1.txt");
  FileInputStream is2 = new FileInputStream("file2.txt");
  SequenceInputStream sis = new SequenceInputStream(is1, is2);
  int b;
  while ((b = sis.read()) != -1)
    System.out.print((char) b);
  sis.close();     // also closes is1, is2
} catch (IOException e) {
  System.err.println("Read error: " + e.getMessage());
}
```

```
First file.
Second file.
```

`file1.txt` contains the line `First file.` and `file2.txt` contains the line `Second file.`. The loop only calls `sis.read()` and does not notice the switch between the files. `sis.close()` closes both inner streams as well. If an error occurs in one of the streams during reading, an `IOException` is thrown and reading does not move on to the other stream.

8.7 Copying files

Copying a file is a good exercise for learning streams: everything read from an input stream is written to an output stream. The same job can be done with different units: byte by byte, block by block, or line by line. This section compares the ways and shows the pitfalls of each.

8.7.1 Copying byte by byte

The simplest copy writes every byte read from the source file to the destination file. Because byte streams do not care about the meaning of the bytes, this method copies every kind of file, whether text, an image, or an `.exe`, without corrupting it. To speed up reading and writing, the two streams are wrapped in buffered streams (Figure 8.17).

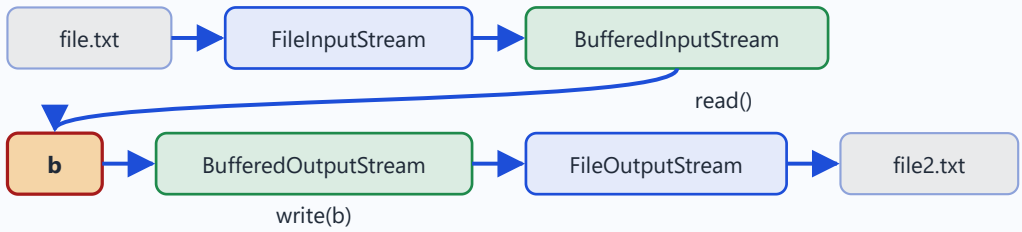


Figure 8.17: The byte-by-byte copy chain. The bytes go from `file.txt` through `FileInputStream` and `BufferedInputStream` into the variable `b`, and from there through `BufferedOutputStream` and `FileOutputStream` to `file2.txt`.

```
import java.io.*;

public class ByteCopy {
  public static void main(String[] args) throws IOException {
    File src = new File("file.txt");
    File dst = new File("file2.txt");
    FileInputStream fis = new FileInputStream(src);
    FileOutputStream fos = new FileOutputStream(dst);
    BufferedInputStream bis = new BufferedInputStream(fis);
    BufferedOutputStream bos = new BufferedOutputStream(fos);
    int b;
    // Read a byte; -1 at end of file.
    while ((b = bis.read()) != -1)
      bos.write(b);    // write it out
    bos.close();
    bis.close();
    System.out.println(dst.length() + " bytes copied");
  }
}
```

The copy loop is a single line: `while ((b = bis.read()) != -1) bos.write(b);`. `bis.read()` reads a byte and returns −1 at the end of the file. `bos.write(b)` writes the byte just read to the output stream. The value read must be kept in an `int` variable. The reason is to tell −1 apart from a byte with the value 255. `read()` can return every value in the range 0–255 as a valid byte; since −1 lies outside this range, it reports the end of the file without ambiguity.

Common mistake

Storing the result of `read()` in a `byte`: `byte b; while ((b = (byte) in.read()) != -1)`. In Java, `byte` is signed and takes values between −128 and 127. A byte with the value 255, that is `FF`, becomes −1 through the `(byte)` cast. The loop mistakes this byte for the end of the file and stops; the rest of the file is not copied. The correct type is `int`.

The effect of the mistake can be seen in a small experiment. If the three-byte array `{10, (byte) 255, 20}` is read with this loop, only the first byte is processed:

```
processed: 1
```

Because the `FF` byte rarely occurs in text files, this mistake goes unnoticed in experiments with text files. In images or compressed files, `FF` is common, and the copy is cut off halfway.

8.7.2 Code trace: byte-by-byte copy

Suppose `file.txt` contains four bytes: `Java`. Table 8.15 gives the state of the two buffers and the destination file at each step of the program.

Step	Line	`b`	`bis` buffer	`bos` buffer	`file2.txt`	What happened?
1	5–6	—	—	—	none	The two `File` objects hold only paths.
2	7–8	—	—	—	0 bytes	`FileOutputStream` creates the destination file; if it exists, its contents are erased.
3	9–10	?	empty	empty	0 bytes	The streams are wrapped in buffered streams.
4	13	74	`Java`	empty	0 bytes	The buffer is empty; one disk read brings 4 bytes. The first byte `'J'` is returned.
5	14	74	`Java`	`J`	0 bytes	`write` puts the byte into the output buffer.
6–7	13–14	97	`Java`	`Ja`	0 bytes	The byte comes from the buffer; the disk is not touched.
8–9	13–14	118	`Java`	`Jav`	0 bytes	The byte comes from the buffer.
10–11	13–14	97	`Java`	`Java`	0 bytes	The output buffer holds 4 bytes.
12	13	−1	empty	`Java`	0 bytes	The buffer is used up; the second disk read reports the end of the file.
13	15	−1	empty	empty	4 bytes	`bos.close()` first does a `flush()`: the 4 bytes go to disk in one write.
14	16	−1	—	—	4 bytes	The input streams are closed.
15	17	−1	—	—	4 bytes	Output: `4 bytes copied`

Table 8.15: Trace of the `ByteCopy` program. The line numbers are the lines of the program; the buffers are 8192 bytes.

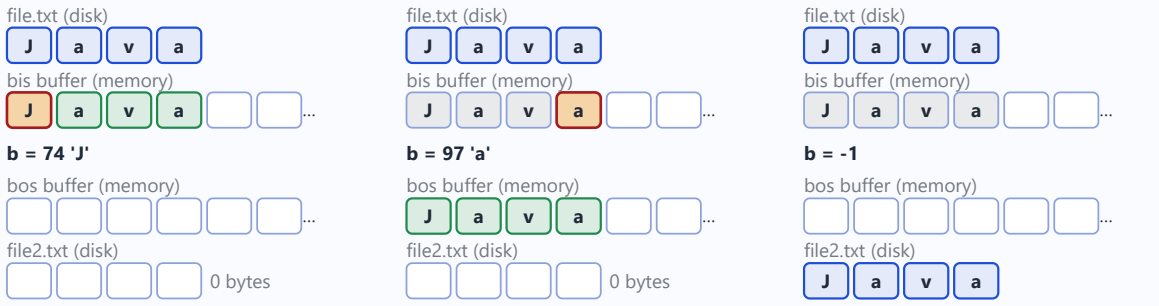


Figure 8.18: Three moments of the byte-by-byte copy. (a) The first `read()` brings all four bytes into the input buffer. (b) Four bytes wait in the output buffer; the destination file is still 0 bytes. (c) After `close()`, the four bytes are on disk.

The most important result of the trace is that although the program makes five `read()` calls and four `write()` calls, the disk is touched far less often. To verify this, subclasses of `FileInputStream` and `FileOutputStream` were written. These subclasses count and print every call that goes to the disk:

```
  disk read(arr, 0, 8192) -> 4
  disk read(arr, 0, 8192) -> -1
  disk write(arr, 0, 4)
buffered: 2 reads, 1 write
  disk read() -> 74
  disk read() -> 97
  disk read() -> 118
  disk read() -> 97
  disk read() -> -1
unbuffered: 5 reads, 4 writes
```

The buffered copy makes two reads from the disk. The first asks for 8192 bytes and gets four. The second happens after the buffer is used up and reports the end of the file (−1). Only one write goes to the disk. The unbuffered copy, in contrast, needs one read and one write for every byte, plus one more read for the end of the file: five reads, four writes. For a four-byte file the difference is small. For a one-megabyte file, the unbuffered copy makes over a million read and write calls. The buffered copy finishes the job with 129 reads and 128 writes, about 260 calls.

8.7.3 Copying block by block

Buffered streams reduce disk access, but the program itself still calls `read()` and `write()` once per byte. The faster way is to read the data in blocks into the program's own array. `read(byte[] buf)` tries to fill the array and returns the number of bytes it read. `write(buf, 0, size)` writes only the first `size` bytes of the array.

```
public class BlockCopy {
  public static void main(String[] args) throws IOException {
    // Data is read in 16K chunks
    byte[] buf = new byte[1024 * 16];
    File src = new File("file.txt");
    File dst = new File("file2.txt");
    FileInputStream fis = new FileInputStream(src);
    FileOutputStream fos = new FileOutputStream(dst);
    BufferedInputStream bis = new BufferedInputStream(fis);
    BufferedOutputStream bos = new BufferedOutputStream(fos);
    int size;
    // Read up to 16K; -1 at end of file.
    while ((size = bis.read(buf)) > -1)
      bos.write(buf, 0, size);   // write the chunk
    bos.close();
    bis.close();
    System.out.println(dst.length() + " bytes copied");
  }
}
```

For a 40,000-byte file the output is:

```
40000 bytes copied
```

`new byte[1024 * 16]` creates a 16 KB array. The data is processed in chunks of this size, so the program's memory use stays constant no matter how large the file is. A 40,000-byte file arrives in three reads: 16,384, 16,384, and 7,232 bytes. The last chunk does not fill the whole array. An experiment with a 40,000-byte file of random bytes, run for these notes, confirms the chunk sizes and that the copy is identical to the source:

```
chunks read: 16384 16384 7232
identical: true
```

The loop condition has the form `> -1`; it means the same as `!= -1`, because `read` never returns a value smaller than −1. A return value of 0 from `read(byte[])` is possible only when the array has length 0.

8.7.4 Code trace: block copy

In block copying, the handling of the last chunk is critical. To keep the trace small, it uses a 16-byte array and a 39-byte file; in real programs the array is 8–16 KB. This time no buffered streams are used, and the resources are opened with try-with-resources.

```
public class BlockTrace {
  public static void main(String[] args) throws IOException {
    byte[] buf = new byte[16];
    try (InputStream in = new FileInputStream("file.txt");
      OutputStream out = new FileOutputStream("file2.txt")) {
      int size;
      while ((size = in.read(buf)) > -1) {
        out.write(buf, 0, size);
        System.out.println("size = " + size);
      }
    }
    long n = new File("file2.txt").length();
    System.out.println("Done: " + n + " bytes");
  }
}
```

Step	Line	Variables	New output
1	3: `byte[] buf = new byte[16];`	buf=byte[16]	—
2	4: `try (InputStream in = new FileInputStream("file.txt");` 5: `OutputStream out = new FileOutputStream("file2.txt")) {`	buf=byte[16]	—
3	7: `while ((size = in.read(buf)) > -1) {`	size=16	—
4	8: `out.write(buf, 0, size);`	size=16	—
5	9: `System.out.println("size = " + size);`	size=16	`size = 16`
6	7: `while ((size = in.read(buf)) > -1) {`	size=16	—
7	8: `out.write(buf, 0, size);`	size=16	—
8	9: `System.out.println("size = " + size);`	size=16	`size = 16`
9	7: `while ((size = in.read(buf)) > -1) {`	size=7	—
10	8: `out.write(buf, 0, size);`	size=7	—
11	9: `System.out.println("size = " + size);`	size=7	`size = 7`
12	7: `while ((size = in.read(buf)) > -1) {`	size=-1	—
13	13: `System.out.println("Done: " + n + " bytes");`	size=-1, n=39	`Done: 39 bytes`

Table 8.16: Trace of the `BlockTrace` program; `file.txt` is 39 bytes.

In the first two iterations of Table 8.16, `read(buf)` fills the array and returns 16. `write(buf, 0, 16)` writes the 16 bytes that were read. In the second iteration the new 16 bytes overwrite the same array; the array is reused in every iteration. In the third iteration only 7 bytes are left in the file. `read` refreshes the first 7 cells of the array and returns 7. The remaining 9 cells still hold bytes left over from the previous block.

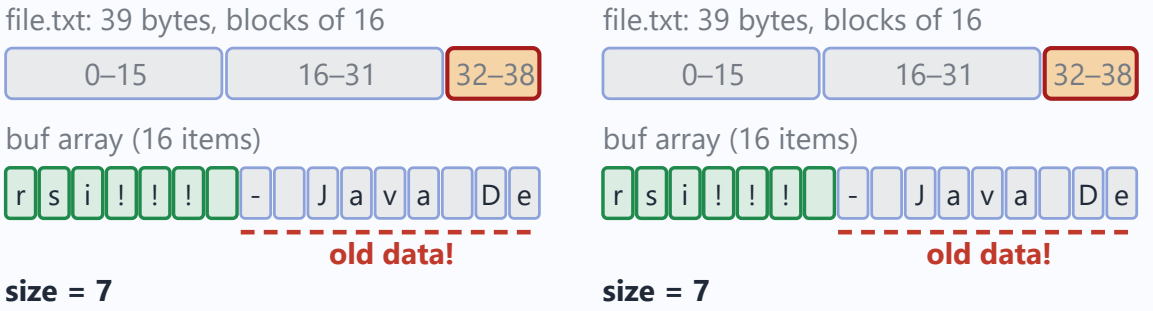


Figure 8.19: The last block. (a) Only 7 bytes are left; the first 7 cells of the array are refreshed, and the remaining 9 cells are left over from the old block. (b) `write(buf, 0, 7)` writes only the 7 bytes that were read.

Had `write(buf)` been written here instead of `write(buf, 0, size)`, all 16 bytes of the array would have been written. The 9 old bytes would also enter the copy, and the copy would be 16 + 16 + 16 = 48 bytes. An experiment with the same file confirms this. The last 9 bytes of the copy are the fragment `kislar ve` left over from the second block:

```
with write(buf): 48 bytes
last 9 bytes: kislar ve | real last block:  tampon
```

In the fourth iteration `read` returns −1, the condition `> -1` becomes false, and the loop ends. The resources are closed, and the copy has the same length as the source: 16 + 16 + 7 = 39 bytes. The number of reads and writes is the ceiling of the file length divided by the array length: ⌈39/16⌉ = 3. A one-megabyte file is copied in 64 iterations with a 16 KB array.

Common mistake

Writing `out.write(buf);` in a block copy. In the last block the array is not completely filled; the stale data at the end of the array also goes into the copy, and the copy becomes larger than the source. The correct call is always `write(buf, 0, size)`, where `size` is the value `read` returned last. The same rule applies when converting to text: `new String(buf, 0, size)`.

8.7.5 Copying line by line

Text files can also be copied line by line. This way works with character streams and is useful when the lines have to be processed while being copied: filtering, numbering, or modifying them, for example.

```
public class LineCopy {
  public static void main(String[] args) throws IOException {
    File src = new File("file.txt");
    File dst = new File("file2.txt");
    FileReader fr = new FileReader(src);
    BufferedReader br = new BufferedReader(fr);
    FileWriter fw = new FileWriter(dst);
    BufferedWriter bw = new BufferedWriter(fw);
    PrintWriter pw = new PrintWriter(bw);
    String line;
    // read a line until end of file
    while ((line = br.readLine()) != null)
      pw.println(line);    // write the line
    pw.close();
    br.close();
    System.out.println("Copied: " + dst.getName());
  }
}
```

```
Copied: file2.txt
```

On the reading side there is `FileReader` and `BufferedReader`, and on the writing side the chain `FileWriter`, `BufferedWriter`, and `PrintWriter`. The loop takes each line with `readLine()` and writes it with `println`. `pw.close()` closes the whole chain, including `bw` and `fw`.

This method has two limits. First, it is suitable only for text files. If a binary file goes through a character stream, the encoding conversion corrupts the bytes. Second, the copy may not be byte-for-byte identical. `readLine()` discards the line terminator, while `println` adds the system's line terminator: `\r\n` on Windows and `\n` on Linux. If the last line of the source has no terminator, the copy has one extra. The experiment below was run on Windows with a 41-byte file whose last line ends without a line terminator; the copy takes 43 bytes:

```
Copied: file2.txt
41 -> 43 bytes; line.separator length 2
```

If the line terminators of the source were in Linux form (`\n`), each line of the copy made on Windows would be one byte longer. So whenever a byte-for-byte identical copy is needed, byte streams are always used.

8.7.6 Downloading a web page

The source of a copy loop does not have to be a file. The following program downloads a web page and saves it to disk. This time the source is the **network**: `url.openStream()` returns an `InputStream`. The rest of the loop is the same as in file copying.

```
public class WebDownload {
  public static void main(String[] args) throws IOException {
    URL url = URI.create("https://sercankulcu.github.io/").toURL();
    BufferedInputStream bis = new BufferedInputStream(url.openStream());
    FileOutputStream fos = new FileOutputStream(new File("copy.html"));
    BufferedOutputStream bos = new BufferedOutputStream(fos);
    for (int c = bis.read(); c != -1; c = bis.read()) {
      bos.write(c);
    }
    bis.close();
    bos.close();
    System.out.println("copy.html: " + new File("copy.html").length() + " bytes");
  }
}
```

```
copy.html: 14283 bytes
```

The size in the output varies with the page and the moment of download. This value is the size of the page as it came from the server when these notes were prepared. This time the copy loop is written with `for`: the first read happens in the initialization part, the byte is written if it is not −1, and the update part reads again. It does the same job as a loop written with `while`.

The use of `URI.create(…).toURL()` in the program is deliberate. The `new URL(String)` constructor has been deprecated since Java 20; this two-step way is recommended instead. `URI` checks the syntax of the address more strictly. If the network connection cannot be made, or the server's certificate cannot be verified, `openStream()` throws a subclass of `IOException`. This program also leaves the error to the JVM with the `throws IOException` in the header of `main`.

8.7.7 Shortcuts and a comparison

There is no need to rewrite the copy loop every time. The Java library has two methods that do this job. `in.transferTo(out)` (Java 9+) transfers all the bytes of an input stream to an output stream and returns the number of bytes transferred. `Files.copy(source, target)` (Java 7+) copies a file directly to another file. Both do block copying internally.

```
public class QuickCopy {
  public static void main(String[] args) throws IOException {
    // Java 9+: transferTo
    try (InputStream in = new FileInputStream("file.txt");
      OutputStream out = new FileOutputStream("file2.txt")) {
      long n = in.transferTo(out);
      System.out.println(n + " bytes transferred");
    }
    // Java 7+: Files.copy
    Files.copy(Path.of("file.txt"), Path.of("file3.txt"),
         StandardCopyOption.REPLACE_EXISTING);
    System.out.println(Files.size(Path.of("file3.txt")) + " bytes copied");
  }
}
```

```
4 bytes transferred
4 bytes copied
```

If the target file exists, `Files.copy` throws `FileAlreadyExistsException` unless overwriting is permitted. This behavior is a safeguard: it prevents an existing file from being overwritten by accident. To overwrite, `StandardCopyOption.REPLACE_EXISTING` is passed as the third argument. An attempt made without the option shows the exception and that the target file is unchanged:

```
java.nio.file.FileAlreadyExistsException: k2.txt
k2.txt: old
```

Table 8.17 compares the copying methods, and Figure 8.20 shows how different the call counts are for a 1 MB file.

Method	Unit	When it fits
Byte by byte (`read()` / `write(int)`)	1 byte	for learning; very slow without buffering
Block by block (`read(byte[])`)	8–16 KB	any kind of file, fast
Line by line (`readLine` / `println`)	1 line	text only; when processing lines
`in.transferTo(out)` (Java 9+)	block	one-line copy between two streams
`Files.copy(source, target)` (Java 7+)	block	shortest way from file to file

Table 8.17: Copying methods.

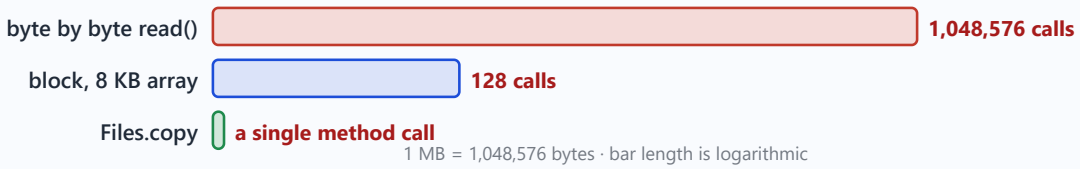


Figure 8.20: Number of `read` calls when copying a 1 MB file (bar lengths are logarithmic). Byte-by-byte reading makes 1,048,576 calls, and block reading with an 8 KB array makes 128; `Files.copy` is a single method call.

Whichever method is chosen, three rules apply: work with buffering, close the streams (preferably with try-with-resources), and handle `IOException`. In a real program, use `Files.copy` for a file-to-file copy and `transferTo` for a copy between two streams. A hand-written block loop is needed when something has to be done with the data while it is copied, for example to show a progress bar or to encrypt the bytes.

8.8 Random access

Streams are all sequential: they read or write data from beginning to end. Some tasks, however, require going directly to a record in the middle of a file. To read the grade of the thousandth student in a student file, reading the previous 999 records is an unnecessary burden.

8.8.1 Sequential and random access

Definition 8.10

In **sequential access**, each read or write continues from where the previous one ended; going back needs special support (for example `mark` / `reset`, within a limited buffer), and you cannot jump freely to a position. In **random access**, the program jumps to a desired position in the file and reads or writes from there. The position is the number of bytes from the start of the file, counting from 0.

Sequential access is like a cassette tape: to reach a song, the earlier ones have to be wound past. Random access is like a vinyl record: the needle is placed wherever you want (Figure 8.21). Streams (`FileInputStream`, `BufferedReader`, and so on) are all sequential. Sequential access is used when the structure of the file is unknown or when the whole file is being processed; copying is the typical example.



Figure 8.21: Sequential access reads the bytes from start to end. Random access jumps to the desired position with `seek`.

Random access can be used when the size and position of the records are known. If all records have the same size, the position of record number k is found by multiplication: position = $k \cdot$ record size (numbering starts at 0). This way is very convenient for reading or modifying a particular record in place. The fixed-size requirement is important. In a text file whose lines have different lengths, where the kth line starts cannot be computed.

8.8.2 `RandomAccessFile`

In Java, random access is done with the class `java.io.RandomAccessFile`. This class differs from streams: a single object both reads and writes, and it is not part of the `InputStream` / `OutputStream` hierarchy. Inside the object there is a **file pointer**. Every read and write takes place at the position the pointer indicates and advances the pointer by the number of bytes read or written. Table 8.18 lists the main members of the class.

Member	Purpose
`RandomAccessFile(String name, String mode)`	mode `"r"`: read only; `"rw"`: read and write (the file is created if it does not exist)
`seek(long pos)`	moves the file pointer to the given position
`getFilePointer()`	distance of the pointer from the start of the file (bytes)
`length()`	length of the file (bytes)
`read()` / `write(int b)`	reads one byte (returned as an `int`) / writes one byte
`readInt()`, `writeInt(int)`, `readDouble()` ...	reads and writes primitive types in binary (`int`: 4 bytes)
`close()`	releases the system resources

Table 8.18: The main members of the `RandomAccessFile` class.

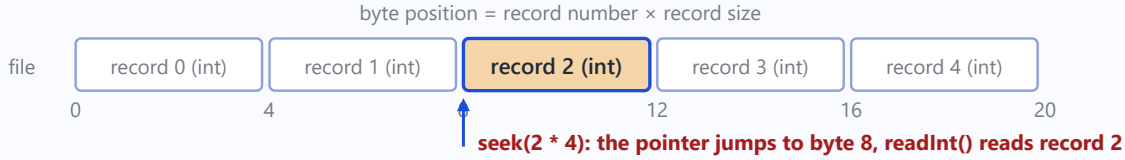


Figure 8.22: 4-byte `int` records. `seek(2 * 4)` moves the pointer to byte 8; `readInt()` reads record number 2. Byte position = record number · record size.

The following program reads the same file first sequentially and then with random access. Here `data` is a 124-byte data file. For these notes the program was run with a file whose first byte is 1 and whose bytes 100–103 are `00 00 01 00`.

```
public class RandomAccess {
  public static void main(String[] args) throws IOException {
    // Sequential access
    FileInputStream seq = new FileInputStream("data");
    int first = seq.read();
    seq.close();
    System.out.println("First byte: " + first);
    // Random access
    RandomAccessFile raf = new RandomAccessFile("data", "rw");
    // Go to 100 bytes from the start
    raf.seek(100);
    // Read a 4-byte int from position 100
    int value = raf.readInt();
    System.out.println("int at 100: " + value);
    raf.close();
  }
}
```

```
First byte: 1
int at 100: 256
```

`FileInputStream` reads from the start of the file and gives the first byte, 1. `seek(100)` moves the pointer to position 100, that is, to **the 101st byte**, because positions start at 0. The four bytes there are `00 00 01 00`. `readInt()` reads the most significant byte first: $0 \cdot 2^{24} + 0 \cdot 2^{16} + 1 \cdot 2^{8} + 0 = 256$.

At the boundaries, `RandomAccessFile` behaves slightly differently from streams. The pointer can be moved beyond the end of the file; if a write is made there, the file grows up to that point and the bytes in between are undefined (they are zero in this experiment). At the end of the file, `read()` returns −1, whereas `readInt()` throws `EOFException`. The mode can only be `"r"`, `"rw"`, `"rws"`, or `"rwd"`; `"w"` is invalid:

```
seek(10)+write: length 11, pointer 11
first byte: 0
read() at the end: -1
readInt() at the end: EOFException
mode r, no such file: FileNotFoundException
mode w: Illegal mode "w" must be one of "r", "rw", "rws", or "rwd"
```

8.8.3 Code trace: random access to records

The following program writes five grades to a file as `int` values. It then reads the third grade directly, modifies the fourth grade in place, and adds up all the grades.

```
import java.io.*;

public class Records {
  public static void main(String[] args) throws IOException {
    RandomAccessFile f = new RandomAccessFile("n.dat", "rw");
    int[] grades = {70, 85, 90, 45, 60};
    for (int n : grades)
      f.writeInt(n);              // 4 bytes each
    System.out.println("Length: " + f.length());
    f.seek(2 * 4);                  // go to record 3
    System.out.println("Grade 3: " + f.readInt());
    System.out.println("Position: " + f.getFilePointer());
    f.seek(3 * 4);                  // record 4
    f.writeInt(50);                 // 50 replaces 45
    f.seek(0);
    int total = 0;
    while (f.getFilePointer() < f.length())
      total += f.readInt();
    System.out.println("Total: " + total);
    f.close();
  }
}
```

Step	Line	Variables	New output
1	5: `RandomAccessFile f = new RandomAccessFile("n.dat", "rw");`	—	—
2	6: `int[] grades = {70, 85, 90, 45, 60};`	grades=[70, 85, 90, 45, 60]	—
3	7: `for (int n : grades)` 8: `f.writeInt(n); // 4 bytes each`	n=70	—
4	7: `for (int n : grades)` 8: `f.writeInt(n); // 4 bytes each`	n=60	—
5	9: `System.out.println("Length: " + f.length());`	—	`Length: 20`
6	10: `f.seek(2 * 4); // go to record 3`	—	—
7	11: `System.out.println("Grade 3: " + f.readInt());`	—	`Grade 3: 90`
8	12: `System.out.println("Position: " + f.getFilePointer());`	—	`Position: 12`
9	13: `f.seek(3 * 4); // record 4`	—	—
10	14: `f.writeInt(50); // 50 replaces 45`	—	—
11	15: `f.seek(0);`	total=0	—

Step	Line	Variables	New output
12	17: `while (f.getFilePointer() < f.length())` 18: `total += f.readInt();`	total=355	—
13	19: `System.out.println("Total: " + total);`	total=355	`Total: 355`

Table 8.19: Trace of the `Records` program.

In the first step of Table 8.19, mode `"rw"` creates the file if it does not exist; the pointer is at 0. In the loop each `writeInt` writes 4 bytes and advances the pointer by 4. When the loop ends, the file is 5 · 4 = 20 bytes and the pointer is at the end of the file, at 20. The actual bytes of the file are:

```
20: 00 00 00 46 00 00 00 55 00 00 00 5A 00 00 00 32 00 00 00 3C
```

`46` = 70, `55` = 85, and `5A` = 90. The fourth record on this line is `32` = 50, because the bytes were printed after the program finished; the value first written was 45, that is, `2D`.

In the sixth step, `seek(2 * 4)` moves the pointer to byte 8. This is the start of the third record, record number 2. The earlier records are not read; the program jumps straight there. `readInt()` reads bytes 8–11 and returns 90. The pointer advances to 12. The `seek(3 * 4)` in the ninth step moves the pointer to 12; the pointer is already there. In the tenth step, `writeInt(50)` is written in place of 45. The rest of the file does not change. This cannot be done with a sequential stream: to change a record in the middle of the file, the whole file would have to be rewritten.

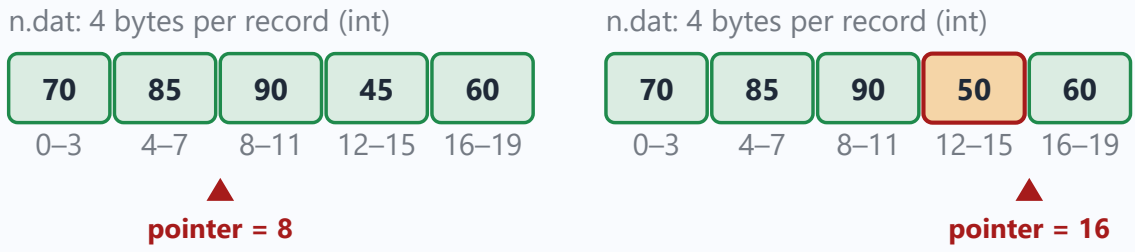


Figure 8.23: Two moments of random access. (a) `seek(8)` moves the pointer to the start of the third record. (b) `writeInt(50)` modifies the fourth record in place.

In the last steps, `seek(0)` returns to the start. The loop reads `int` values until the pointer reaches the file length: 70 + 85 + 90 + 50 + 60 = 355. The program's output is the same as the values on the slides.

```
Length: 20
Grade 3: 90
Position: 12
Total: 355
```

In terms of cost, the difference is clear. To reach record number k with a sequential stream, the preceding k records must be read; the time is proportional to k, $O(k)$. With random access the position is computed by a multiplication and a single `seek` is made; the time is independent of the record number, $O(1)$.

Example 8.1

In a student file, let each record consist, in order, of an `int` student number, a `double` average, and an `int` credit count. Records are numbered from 0. How is the average of record 37 read?

The record size is 4 + 8 + 4 = 16 bytes. Record 37 starts at byte $37 \cdot 16 = 592$. Within the record, the average comes after the student number, 4 bytes past the start of the record. Write `f.seek(37 * 16 + 4); double avg = f.readDouble();` . After the read, the pointer is at 592 + 4 + 8 = 604, which is the start of the credit field of the same record. If one of the records had been written with `float` instead of `double` , all the position calculations would break. Random access requires the file format to be known exactly.

8.9 Serialization

So far numbers and text have been written to files. In an object-oriented program, however, what we want to store is often an object: the state of a game, a student record, a settings object. Writing and reading back every field of an object one by one is tedious. Java offers a mechanism that does this in a single call.

8.9.1 What is serialization?

Definition 8.11

Serialization is converting the state of an object, that is, the values of its fields, into a sequence of bytes. This byte sequence can be written to a file or carried to another environment over a network. Rebuilding an object from a byte sequence is called **deserialization**.

For the objects of a class to be serializable, the class must implement the `java.io.Serializable` interface. This interface has no methods at all. Such interfaces are called **marker interfaces**; their only job is to mark the class as "serializable". The serialization mechanism looks for this mark before writing the object.

Serialization involves three flows (Figure 8.24). An object sits in memory. `ObjectOutputStream.writeObject` converts it to bytes; the file begins with the magic number `AC ED`. `ObjectInputStream.readObject` builds a new object from these bytes. The new object has the same values but is a different object.

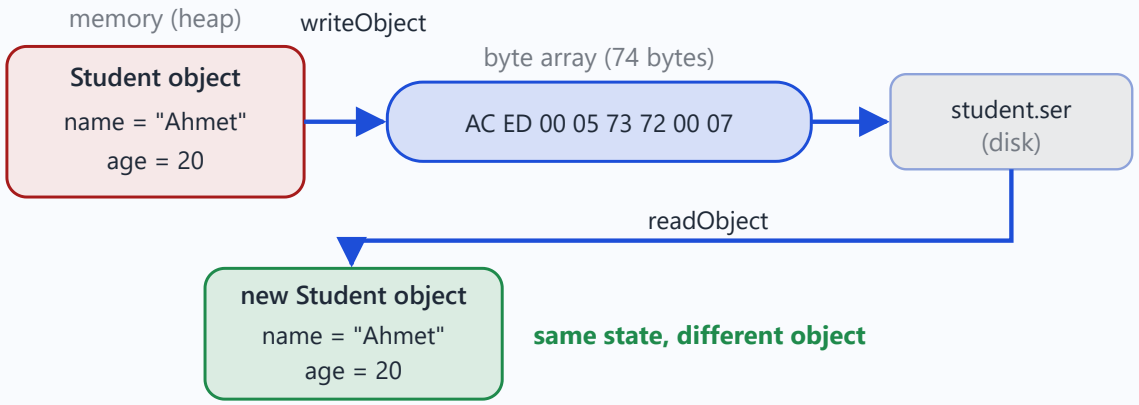


Figure 8.24: Serialization and deserialization. `writeObject` converts the `Student` object in memory into a 74-byte array and writes it to the file; `readObject` builds an object with the same state but a different identity.

Serialization is used in game saves, in caches, and in sending objects over a network to another JVM. Its format is specific to Java and tightly bound to the structure of the class. For this reason, text formats such as JSON are generally preferred for data that will be stored for a long time or shared with other languages. Moreover, deserializing data that comes from an untrusted source can create a security hole, because during deserialization the program's classes are instantiated with unknown data.

8.9.2 A serializable class

```
import java.io.Serializable;

class Student implements Serializable {
  private static final long serialVersionUID = 1L;
  private String name;
  private int age;

  public Student(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public String toString() {
    return "Student [name=" + name + ", age=" + age + "]";
  }
}
```

Three details in the class are related to serialization:

- `implements Serializable` makes the class serializable. No method has to be written.
- `serialVersionUID` is the **version identifier** of the class. This number is also written into the serialized bytes. When reading, the number in the file is compared with the number of the class in the program; if they differ, `InvalidClassException` is thrown. Writing the number by hand is recommended. If it is not written, the serialization mechanism computes it from the structure of the class; when a field or method is added to the class, the computed number changes and old files can no longer be read.
- The fields must be serializable too. `String` and `int` are serializable. If the type of a field is not `Serializable`, `NotSerializableException` is thrown during writing.

The version check really works. When a file written with `serialVersionUID = 1L` is read with version `2L` of the same class, this exception results:

```
java.io.InvalidClassException: Student; local class incompatible: stream classdesc
serialVersionUID = 1, local class serialVersionUID = 2
```

8.9.3 Writing an object and reading it back

`ObjectOutputStream` is used to serialize and `ObjectInputStream` to deserialize. Both are wrapper streams and wrap a file stream.

```
// Serialize
try (ObjectOutputStream out = new ObjectOutputStream(
    new FileOutputStream("student.ser"))) {
  Student student = new Student("Ahmet", 20);
  out.writeObject(student);
  System.out.println("Serialization succeeded.");
} catch (IOException e) {
  e.printStackTrace();
}
System.out.println("student.ser: " + new File("student.ser").length() + " bytes");
// Deserialize
try (ObjectInputStream in = new ObjectInputStream(
    new FileInputStream("student.ser"))) {
  Student student = (Student) in.readObject();
  System.out.println("Student: " + student);
} catch (IOException | ClassNotFoundException e) {
  e.printStackTrace();
}
```

```
Serialization succeeded.
student.ser: 74 bytes
Student: Student [name=Ahmet, age=20]
```

The first block creates a `Student` object and writes it with `writeObject`. The file is 74 bytes. That seems like a lot for this small object. But the file holds not only the values of the two fields but also the definition of the class: the class name, the version identifier, and the names and types of the fields. The bytes of the file show this:

```
AC ED 00 05 73 72 00 07 |
····sr··Student··········I··ageL··namet··Ljava/lang/String;xp····t··Ahmet
```

The first four bytes `AC ED 00 05` are the stream header: the magic number and the version of the serialization protocol. In the readable part you can see `Student`, `age`, `name`, `java/lang/String`, and `Ahmet`.

The second block reads the object back. `readObject` returns a value of type `Object`; that is why the cast `(Student)` is needed. `readObject` can throw two checked exceptions: `IOException` and `ClassNotFoundException`. The second is thrown when the class in the file cannot be found in the program. The program catches both together with a multi-catch. If `ClassNotFoundException` is neither caught nor declared, the code does not compile:

```
Main.java:6: error: unreported exception ClassNotFoundException; must be caught or
declared to be thrown
      Object o = g.readObject();
                             ^
1 error
```

If the cast is made to the wrong type, `ClassCastException` is thrown at run time; if the file is not serialized data at all, `StreamCorruptedException` is thrown:

```
ClassCastException
StreamCorruptedException: invalid stream header: 6D657268
```

`6D657268` is the first four bytes of a plain text file: `m`, `e`, `r`, `h`. Because the stream header `AC ED` is missing, the read is rejected right at the start.

8.9.4 `transient` and `static` fields

Not every field of an object should be stored. A password should not be stored. A temporary cache or an open file connection cannot even be stored. Such fields are marked with the keyword `transient`.

Definition 8.12

A field marked `transient` is skipped during serialization. When the object is read back, this field gets the default value of its type: `null` for references, 0 for numbers, and `false` for `boolean`. `static` fields are never serialized, because they belong to the class, not to the object.

```
class Account implements Serializable {
  private static final long serialVersionUID = 1L;
  static int count;
  private String name;
  private transient String password;   // not written

  Account(String name, String password) {
    this.name = name;
    this.password = password;
  }

  public String toString() {
    return "Account [name=" + name + ", password=" + password + "]";
  }
}
```

The experiment below writes an `Account` object, changes the `static` counter, and reads the object back:

```
Account k = new Account("ayse", "secret12");
Account.count = 5;
ObjectOutputStream c = new ObjectOutputStream(
    new FileOutputStream("k.ser"));
c.writeObject(k);
c.close();
Account.count = 99;        // static: per class
ObjectInputStream g = new ObjectInputStream(
    new FileInputStream("k.ser"));
Account copy = (Account) g.readObject();
g.close();
System.out.println(copy);
System.out.println("count = " + Account.count);
System.out.println("same object? " + (copy == k));
```

```
Account [name=ayse, password=null]
count = 99
same object? false
```

Each of the three lines shows one rule. `password` came back as `null`, because the `transient` field was not written to the file. `count` is 99, because the file does not contain `count` and reading cannot change it; the value is the last one in memory. `copy == k` is `false`, because `readObject` builds a new object.

8.9.5 Code trace: serialization

When the same piece of code is traced step by step, the size of the file and the state of the objects can be seen at every moment. The trace program added two lines to the code on the slides so that it also prints the file size:

```
[after header: 4]
[k.ser: 63]
Account [name=ayse, password=null]
count = 99
same object? false
```

Step	Line	Variables	New output
1	1: `Account k = new Account("ayse", "secret12");`	k=Account@1	—
2	2: `Account.count = 5;`	k=Account@1, Account.count=5	—
3	3: `ObjectOutputStream c = new ObjectOutputStream(` 4: `new FileOutputStream("k.ser"));`	c=ObjectOutputStream	—
4	5: `c.writeObject(k);`	c=ObjectOutputStream	—
5	6: `c.close();`	c=closed	—
6	7: `Account.count = 99; // static: per class`	Account.count=99	—
7	8: `ObjectInputStream g = new ObjectInputStream(` 9: `new FileInputStream("k.ser"));`	g=ObjectInputStream	—
8	10: `Account copy = (Account) g.readObject();`	copy=Account@2	—
9	12: `System.out.println(copy);`	copy=Account@2	`Account [name=ayse, password=null]`
10	13: `System.out.println("count = " + Account.count);`	Account.count=99	`count = 99`
11	14: `System.out.println("same object? " + (copy == k));`	result of copy == k: false	`same object? false`

Table 8.20: Trace of serialization. `Account@1` and `Account@2` denote two separate objects on the heap.

In the first step of Table 8.20, a new `Account` object is created on the heap; `name` and `password` are filled in. The `static` `count` is a field of the class and has a single copy. In the third step the `ObjectOutputStream` is opened and immediately writes the 4-byte stream header to the file. In the fourth step `writeObject` writes the class definition and the fields; the file becomes 63 bytes. The `transient` `password` and the `static` `count` are not written.

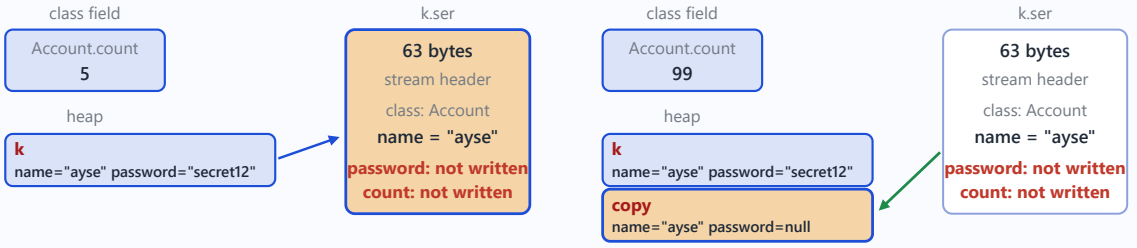


Figure 8.25: Two moments from the serialization trace. (a) `writeObject` writes only `name`; the file is 63 bytes. (b) `readObject` builds a new object; `password` is `null`.

The eighth step carries a subtle point of serialization. `readObject` builds a new object on the heap but **does not call the constructor** of `Account`. The values of the fields come directly from the file. Fields that are not in the file keep their default values. This is why the initial value in the declaration of a `transient` field does not take effect either: even if `transient int b = 2;` is written, `b` is 0 in the object read back. In the last step two references are compared; although their contents are similar, they are two separate objects.

Common mistake

Forgetting `Serializable`. If the class written with `writeObject`, or the type of one of its fields, does not implement this interface, an exception such as `java.io.NotSerializableException: Car` is thrown at run time. The compiler does not check this with default settings. The fix is to make the class `Serializable` or to mark the field `transient`.

8.10 NIO

Java NIO (New I/O) arrived with Java 1.4 in the `java.nio` package. It offers higher performance and flexibility than the classic `java.io` streams. NIO.2 (`java.nio.file`), which came with Java 7, added the `Path` and `Files` classes for file operations.

8.10.1 The building blocks of NIO

NIO is built on four basic concepts:

- **Buffer** (`Buffer`): memory areas that hold data temporarily.
- **Channel** (`Channel`): connection objects that manage input and output operations and can be bidirectional.
- **Selector** (`Selector`): an object that monitors multiple channels with a single thread and reports which are ready.
- **Path** (`Path`): the NIO.2 class that represents file and directory paths.

8.10.2 Stream or channel?

In classic I/O the program works with a stream, and data flows in one direction, byte by byte or character by character. In NIO the program writes to a buffer or reads from a buffer, and a channel moves whole buffers of data between the buffer and the source or destination (Figure 8.26).

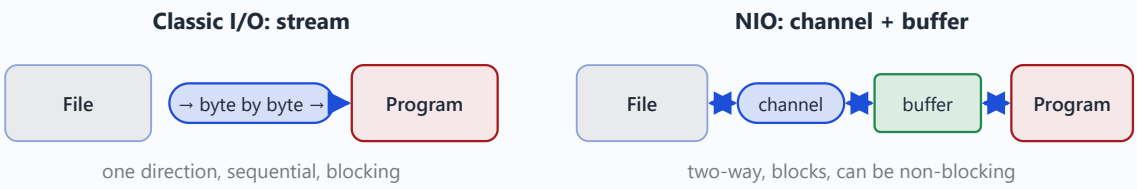


Figure 8.26: A classic stream is one-way and sequential. In NIO the program transfers blocks in both directions through a channel and a buffer.

	Classic I/O (`java.io`)	NIO (`java.nio`)
Core concept	stream	channel and buffer
Direction	one way: input or output	a channel can be two-way
Unit	byte or character	a block of a buffer's size
Waiting	always blocking	a non-blocking mode exists too
Many connections	one thread per connection	a single thread and a selector
When to use	simple file and console tasks	servers with many connections, large files

Table 8.21: Classic I/O compared with NIO.

The non-blocking mode is the most important difference of NIO. With a classic stream, `read()` waits until data arrives, and the thread cannot do anything else in the meantime. A server with a thousand clients has to start a thousand threads. A non-blocking channel, by contrast, returns immediately if there is no data. A single thread can therefore poll many channels in turn.

8.10.3 Buffers and their properties

Definition 8.13

An NIO **buffer** is a fixed-size block of memory that holds data temporarily: an array plus position information over it. It has four properties. The **capacity** is the maximum number of elements it can hold and never changes. The **position** is the index at which the next read or write takes place. The **limit** is the first index that cannot be read or written. The **mark** is a position saved with `mark()`; `reset()` returns to it. Always 0 ≤ mark ≤ position ≤ limit ≤ capacity.

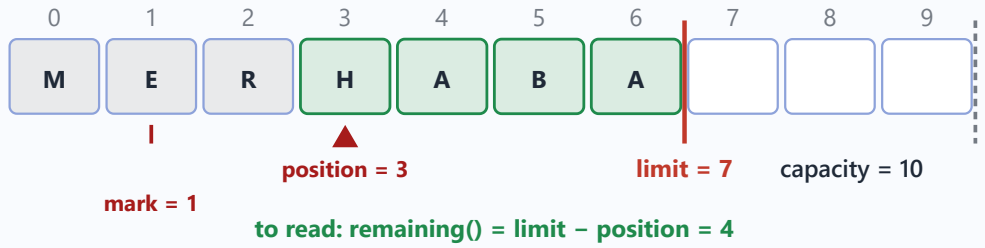


Figure 8.27: A buffer with capacity 10. The position is 3, the limit 7, and the mark 1. The number of elements left to read is `remaining()` = limit − position = 4.

There is a buffer class for each primitive type: `ByteBuffer`, `CharBuffer`, `ShortBuffer`, `IntBuffer`, `LongBuffer`, `FloatBuffer`, and `DoubleBuffer`. Because channels carry bytes, `ByteBuffer` is used most. A buffer is created either with `ByteBuffer.allocate(1024)` or by wrapping an existing array with `ByteBuffer.wrap(array)`.

A buffer has two modes. In **write mode** the position points to the next empty cell and the limit to the capacity. In **read mode** the position points to the next cell to read and the limit to the end of the data written. The modes are switched with `flip()`. The buffer does not know which mode it is in; the programmer has to keep track of it.

8.10.4 `ByteBuffer` examples

```
public class BufferDemo {
  public static void main(String[] args) {
    // Create a 1 KB ByteBuffer
    ByteBuffer buffer = ByteBuffer.allocate(1024);
    // Write data
    buffer.put("Hello from NIO".getBytes());
    System.out.println("after put  : " + buffer);
    // Switch to read mode
    buffer.flip();
    System.out.println("after flip : " + buffer);
    // Read the data
    while (buffer.hasRemaining()) {
      System.out.print((char) buffer.get());
    }
    System.out.println();
  }
}
```

```
after put  : java.nio.HeapByteBuffer[pos=14 lim=1024 cap=1024]
after flip : java.nio.HeapByteBuffer[pos=0 lim=14 cap=1024]
Hello from NIO
```

`put` writes 14 bytes and advances the position to 14. `flip()` pulls the limit down to 14 and returns the position to 0; the buffer is now in read mode. `hasRemaining()` asks whether the position is smaller than the limit. The loop reads exactly 14 bytes. Had `flip()` been forgotten, reading would start at position 14 and 1010 empty bytes would be read.

This loop gives the right result only for ASCII text. Converting each byte to a `char` on its own corrupts multibyte characters:

```
public class BufferUmlaut {
  public static void main(String[] args) {
    ByteBuffer b = ByteBuffer.allocate(64);
    b.put("Dünya".getBytes(StandardCharsets.UTF_8));
    b.flip();
    while (b.hasRemaining())
      System.out.print((char) b.get());      // wrong
    System.out.println();
    b.rewind();
    System.out.println(StandardCharsets.UTF_8.decode(b));  // right
  }
}
```

```
D╫ᴇnya
Dünya
```

"Dünya" is 6 bytes in UTF-8; `ü` = `C3 BC`. `(char) b.get()` converts a signed `byte` value to `char`. `C3` as a `byte` is −61, and when it is converted to `char` the sign bit is extended: `U+FFC3`, that is, `╫`. `BC` likewise becomes `ᴇ`. To decode the text, use `Charset.decode`. Because `rewind()` brings the position back to 0, the same bytes are decoded correctly this time.

8.10.5 Code trace: buffer operations

The following program uses six buffer operations in turn: `put`, `flip`, `get`, `compact`, `rewind`, and `clear`.

```
import java.nio.ByteBuffer;

public class BufferTrace {
  public static void main(String[] args) {
    ByteBuffer b = ByteBuffer.allocate(8);
    b.put((byte) 'J');
    b.put((byte) 'A');
    b.put((byte) 'V');
    b.put((byte) 'A');
    b.flip();
    char c1 = (char) b.get();
    char c2 = (char) b.get();
    b.compact();
    b.put((byte) '!');
    b.flip();
    b.get();
    b.rewind();
    b.clear();
    System.out.println(c1 + "" + c2 + " " + b);
  }
}
```

Step	Line	Variables	New output
1	5: `ByteBuffer b = ByteBuffer.allocate(8);`	pos=0, lim=8, cap=8	—
2	6: `b.put((byte) 'J');`	pos=1, lim=8, cap=8	—
3	7: `b.put((byte) 'A');`	pos=2, lim=8, cap=8	—
4	8: `b.put((byte) 'V');`	pos=3, lim=8, cap=8	—
5	9: `b.put((byte) 'A');`	pos=4, lim=8, cap=8	—
6	10: `b.flip();`	pos=0, lim=4, cap=8	—

Step	Line	Variables	New output
7	11: `char c1 = (char) b.get();`	pos=1, lim=4, cap=8, c1='J'	—
8	12: `char c2 = (char) b.get();`	pos=2, lim=4, cap=8, c1='J', c2='A'	—
9	13: `b.compact();`	pos=2, lim=8, cap=8, c1='J', c2='A'	—
10	14: `b.put((byte) '!');`	pos=3, lim=8, cap=8, c1='J', c2='A'	—
11	15: `b.flip();`	pos=0, lim=3, cap=8, c1='J', c2='A'	—
12	16: `b.get();`	pos=1, lim=3, cap=8, c1='J', c2='A'	—
13	17: `b.rewind();`	pos=0, lim=3, cap=8, c1='J', c2='A'	—
14	18: `b.clear();`	pos=0, lim=8, cap=8, c1='J', c2='A'	—
15	19: `System.out.println(c1 + "" + c2 + " " + b);`	pos=0, lim=8, cap=8, c1='J', c2='A'	`JA java.nio.HeapByteBuffer[pos=0 lim=8 cap=8]`

Table 8.22: Trace of the buffer operations; `pos` is the position, `lim` the limit, and `cap` the capacity.

The variables in the table do not show the inside of the buffer. A tracing program that prints all eight cells of the buffer after every operation fills this gap (`·` is an empty cell):

```
allocate   pos=0 lim=8 [········]
put J      pos=1 lim=8 [J·······]
put A      pos=2 lim=8 [JA······]
put V      pos=3 lim=8 [JAV·····]
put A      pos=4 lim=8 [JAVA····]
flip       pos=0 lim=4 [JAVA····]
get J      pos=1 lim=4 [JAVA····]
get A      pos=2 lim=4 [JAVA····]
compact    pos=2 lim=8 [VAVA····]
put !      pos=3 lim=8 [VA!A····]
flip       pos=0 lim=3 [VA!A····]
get V      pos=1 lim=3 [VA!A····]
rewind     pos=0 lim=3 [VA!A····]
clear      pos=0 lim=8 [VA!A····]
```

In the first five steps, four `put` calls write `JAVA` and the position advances to 4. `flip()` pulls the limit down to the old position, 4, and the position becomes 0. Now only the four bytes written can be read. Two `get` calls read `J` and `A`; two unread bytes remain.

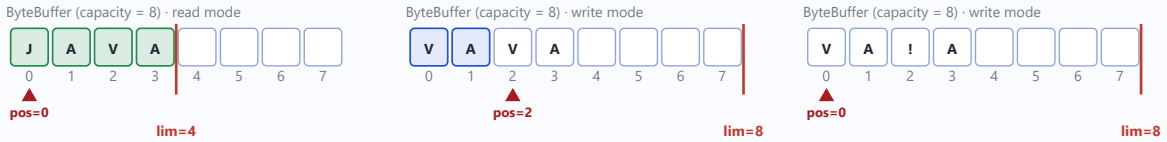


Figure 8.28: Three moments in the buffer operations. (a) `flip()`: limit 4, position 0. (b) `compact()`: the unread `V` and `A` are moved to the front, position 2, limit 8. (c) `clear()`: position 0, limit 8; the data is not erased.

`compact()` copies the unread `V` and `A` to the beginning of the buffer. The position is set to the number of remaining elements, 2, and the limit to the capacity, 8; the buffer switches back to write mode. One point deserves attention: the old `V` and `A` in cells 2 and 3 are not erased. The cell contents are `VAVA`; newly written data overwrites these old bytes. The next `put('!')` overwrites the old `V` at position 2: `VA!A`.

The second `flip()` pulls the limit down to 3; the readable data is `V`, `A`, `!`. One `get` reads `V`. `rewind()` returns the position to 0 but does not change the limit; the same data can be read again from the start. `clear()` sets the position to 0 and the limit to the capacity. Despite its name, it does not erase the data; the buffer is merely considered "empty" and gets overwritten. The last line shows that `toString()` reports the position, limit, and capacity: `JA java.nio.HeapByteBuffer[pos=0 lim=8 cap=8]`. Table 8.23 summarizes the operations.

Method	What it does	Position	Limit
`put(x)`	writes data at the position	+1	—
`get()`	reads the data at the position	+1	—
`get(i)`	reads the data at an absolute index	unchanged	—
`flip()`	switches from writing to reading	0	old position
`rewind()`	to read the same data again from the start	0	unchanged
`clear()`	prepares for writing again; does not erase the data	0	capacity
`compact()`	moves the unread data to the front and continues writing	number remaining	capacity
`mark()` / `reset()`	marks the position / returns to the mark	— / mark	—

Table 8.23: The effect of buffer operations on the position and the limit.

The advantage of a buffer is that the same block of memory is used again and again. The data can be navigated and modified flexibly. The drawback is that it is complicated to use: forgetting `flip` is the most common mistake. The size of a buffer cannot be changed afterwards; writing past the limit throws `BufferOverflowException`, and reading past the limit throws `BufferUnderflowException`:

```
java.nio.HeapByteBuffer[pos=0 lim=5 cap=5]
BC then reset: pos=1 get=B
BufferOverflowException
BufferUnderflowException
java.nio.HeapIntBuffer[pos=0 lim=2 cap=3] 7 8
putInt+putShort pos=6
```

The second line of this output shows `mark` and `reset`: the mark is set while the position is 1, and after two reads `reset()` brings the position back to 1. In the last line, `putInt` and `putShort` advance the position by 4 + 2 = 6. Like `DataOutputStream`, `ByteBuffer` can write primitive types in binary.

Common mistake

Forgetting `flip()`. If `b.get()` is called directly after `b.put((byte) 'J'); b.put((byte) 'A');`, the empty cell at position 2 is read and 0 is returned. What is read is not the data written but whatever lies after it. `flip()` is required after writing and before reading; so is `clear()` or `compact()` before writing again. There is also a type error: the `put` method of `ByteBuffer` wants a `byte`, and `b.put('J')` does not compile, because a `char` is not converted to a `byte` automatically.

8.10.6 Channels

Definition 8.14

A **channel** is a connection that exchanges data with a file, a network socket, or another I/O device. Data from a channel is always read into a buffer (`read(buffer)`) and written from a buffer (`write(buffer)`).

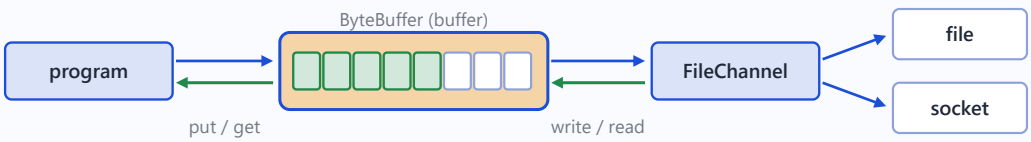


Figure 8.29: The program does `put` into the buffer and `get` from the buffer; the channel writes the buffer to a file or a socket and reads from them into the buffer.

There are four basic channel types: `FileChannel` reads and writes files, `SocketChannel` transmits data over a TCP network socket, `ServerSocketChannel` accepts incoming TCP connections, and `DatagramChannel` transmits data over a UDP socket. Channels are opened with factory methods such

as `FileChannel.open(path, options…)` and `SocketChannel.open()`. When the work is done, they are closed with `close()` or with try-with-resources.

```
public class ChannelOpenClose {
  public static void main(String[] args) throws IOException {
    // Create a FileChannel
    FileChannel fileChannel = FileChannel.open(
        Paths.get("file.txt"), StandardOpenOption.READ);
    // Create a SocketChannel
    SocketChannel socketChannel = SocketChannel.open();
    socketChannel.connect(new InetSocketAddress("example.com", 80));
    System.out.println("Connected: " + socketChannel.isConnected());
    // Close the channels
    fileChannel.close();
    socketChannel.close();
  }
}
```

```
Connected: true
```

This program opens two channels: a file channel and a network channel connected to port 80 of `example.com`. In blocking mode, `connect` waits until the connection is established. The program was actually run for these notes, and the connection succeeded. Two `close()` calls keep the channels from being left open; if an exception occurs, these lines are never reached. That is why try-with-resources is safer.

Reading with a file channel follows the typical pattern of buffer use:

```
public class ChannelRead {
  public static void main(String[] args) throws IOException {
    try (FileChannel channel = FileChannel.open(
        Paths.get("file.txt"), StandardOpenOption.READ)) {
      ByteBuffer buffer = ByteBuffer.allocate(1024);
      // Read from the file
      int bytesRead = channel.read(buffer);
      System.out.println(bytesRead + " bytes read");
      // Switch to read mode
      buffer.flip();
      // Print the data
      while (buffer.hasRemaining()) {
        System.out.print((char) buffer.get());
      }
    }
  }
}
```

```
17 bytes read
NIO channel demo
```

`file.txt` contains `NIO channel demo` and a line terminator: 16 + 1 = 17 bytes. `channel.read(buffer)` reads from the channel into the buffer and returns the number of bytes read; at the end of the file it returns −1. Reading is a *write* into the buffer; `flip()` is needed before reading from the buffer. In a large file a single `read` cannot read the whole file; the reading is then done in a loop, and at the end of each round the buffer is emptied with `clear()` or `compact()`.

```
public class ChannelWrite {
  public static void main(String[] args) throws IOException {
    try (FileChannel ch = FileChannel.open(Paths.get("out.txt"),
        StandardOpenOption.CREATE, StandardOpenOption.WRITE)) {
      byte[] data = "Written via channel\n".getBytes(StandardCharsets.UTF_8);
      ByteBuffer buf = ByteBuffer.wrap(data);
      int written = ch.write(buf);
      System.out.println(written + " bytes written");
    }
  }
}
```

```
20 bytes written
```

`ByteBuffer.wrap(data)` uses a ready-made array as a buffer. The position is 0 and the limit is the size of the array; since the buffer is already in read mode, no `flip()` is needed. The text is 20 bytes in UTF-8: 19 characters plus the line terminator. The `CREATE` option creates the file if it does not exist. If the file exists and is longer, `CREATE` with `WRITE` does not shorten it; the new bytes are written at the start and the end of the old content remains. To reset the file, add `TRUNCATE_EXISTING`.

A file channel is two-way. The same channel can both read and write, and it can go to any position with `position`; in this respect it resembles `RandomAccessFile`:

```
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.file.*;
public class ChannelReadWrite {
  public static void main(String[] args) throws IOException {
    try (FileChannel ch = FileChannel.open(Path.of("both.dat"), StandardOpenOption.CREATE,
        StandardOpenOption.READ, StandardOpenOption.WRITE,
StandardOpenOption.TRUNCATE_EXISTING)) {
      ch.write(ByteBuffer.wrap("ABCDEF".getBytes()));
      System.out.println("position after writing " + ch.position() + ", size " +
ch.size());
      ch.position(2);
      ByteBuffer t = ByteBuffer.allocate(3);
      int n = ch.read(t);
      t.flip();
      System.out.println(n + " bytes: " + new String(t.array(), 0, t.limit()));
      ByteBuffer u = ByteBuffer.allocate(10);
      System.out.println("next read: " + ch.read(u));
    }
  }
}
```

```
position after writing 6, size 6
3 bytes: CDE
next read: 1
```

A network channel works with the same pattern. The following fragment is the body of a `main` method: it sends an HTTP request to a web server and prints the first line of the response.

```
try (SocketChannel socketChannel = SocketChannel.open()) {
  socketChannel.connect(new InetSocketAddress("example.com", 80));
  ByteBuffer buffer = ByteBuffer.allocate(1024);
  buffer.put(("GET / HTTP/1.1\r\nHost: example.com\r\n"
      + "Connection: close\r\n\r\n").getBytes(StandardCharsets.US_ASCII));
  buffer.flip();
  // Write the request to the socket
  socketChannel.write(buffer);
  // Read the response
  buffer.clear();
  socketChannel.read(buffer);
  buffer.flip();
  String reply = StandardCharsets.US_ASCII.decode(buffer).toString();
  System.out.println(reply.substring(0, reply.indexOf("\r\n")));
} catch (IOException e) {
  e.printStackTrace();
}
```

```
HTTP/1.1 200 OK
```

The request is first put into the buffer, then flipped and written to the channel. For the response, the buffer is emptied with `clear()`, read from the channel, and flipped again. The `Host` header is mandatory in HTTP/1.1 requests. The first line of the response is `HTTP/1.1 200 OK`; this line is the program's real output from when these notes were prepared. A single `read` call may not receive the whole response. A real client keeps reading until the server closes the connection and `read` returns −1.

8.10.7 `Path` and `Files`

The `Path` class of NIO.2 is the modern counterpart of `File`. The `Files` class contains `static` methods that reduce common file tasks to a single call.

```
public class FilesNio {
  public static void main(String[] args) throws IOException {
    Path path = Path.of("data", "grades.txt");      // Java 11+
    Files.createDirectories(path.getParent());
    Files.writeString(path, "Ali 70\nAyşe 85\nCan 90\n");
    System.out.println(path.getFileName() + ": " + Files.exists(path));
    System.out.println("Size: " + Files.size(path) + " bytes");
    List<String> lines = Files.readAllLines(path);
    for (String line : lines) {
      System.out.println(line.toUpperCase(Locale.ROOT));
    }
    Files.write(path, List.of("Deniz 60"), StandardOpenOption.APPEND);
    System.out.println(Files.readAllLines(path).size() + " lines");
  }
}
```

```
grades.txt: true
Size: 23 bytes
ALI 70
AYŞE 85
CAN 90
4 lines
```

`Path.of("data", "grades.txt")` builds the path from parts and inserts the system's separator itself. `createDirectories` creates the directory together with any intermediate directories. `writeString` writes the text in UTF-8 in a single call: since the `ş` in `Ayşe` is two bytes, 22 characters take 23 bytes. `readAllLines` reads the whole file as a list of lines. With `Files.write`, the `APPEND` option adds a new line at the end, and the file has 4 lines.

The call `toUpperCase(Locale.ROOT)` in the program is deliberate. When `toUpperCase()` is called without a parameter, it uses the computer's locale. In the Turkish locale, the uppercase of a lowercase `i` is `İ`; on a Turkish system the program would print `ALİ 70`, and on an English system `ALI 70`. `Locale.ROOT` (`java.util.Locale`) makes the conversion language-independent, so the output is the same on every machine. This form is used for conversions that must not depend on the language, for example for file names or keywords. The difference looks like this in the Turkish locale:

```
tr: ALİ 70 / ROOT: ALI 70 / tıtle
```

Method	Purpose
`Path.of(…)` / `Paths.get(…)`	creates a path (the modern counterpart of `File`)
`Files.readAllLines`, `Files.readString`	reads a whole file in one call (UTF-8)
`Files.write`, `Files.writeString`	writes in one call
`Files.exists`, `size`, `copy`, `move`, `delete`	file system operations

Table 8.24: Frequently used methods of `Path` and `Files`.

The methods of `Files` are safer than those of `File` in one respect: they report failure not by returning `false` but with a descriptive exception. If `Files.delete` cannot delete a non-empty directory, it throws `DirectoryNotEmptyException`. Because `readAllLines` and `readString` load the whole file into memory, they are meant for small and medium-sized files; very large files are read line by line with `BufferedReader`.

8.10.8 The selector

Definition 8.15

A **selector** is an NIO object that watches several channels from a single thread. Channels are put into non-blocking mode and registered with the selector. `select()` waits until at least one channel is ready and returns the number of ready channels.

A selector is created with `Selector.open()`. A channel is put into non-blocking mode with `configureBlocking(false)` and registered with `channel.register(selector, events)`. Which events the channel waits for is specified with `SelectionKey` constants (Table 8.25).

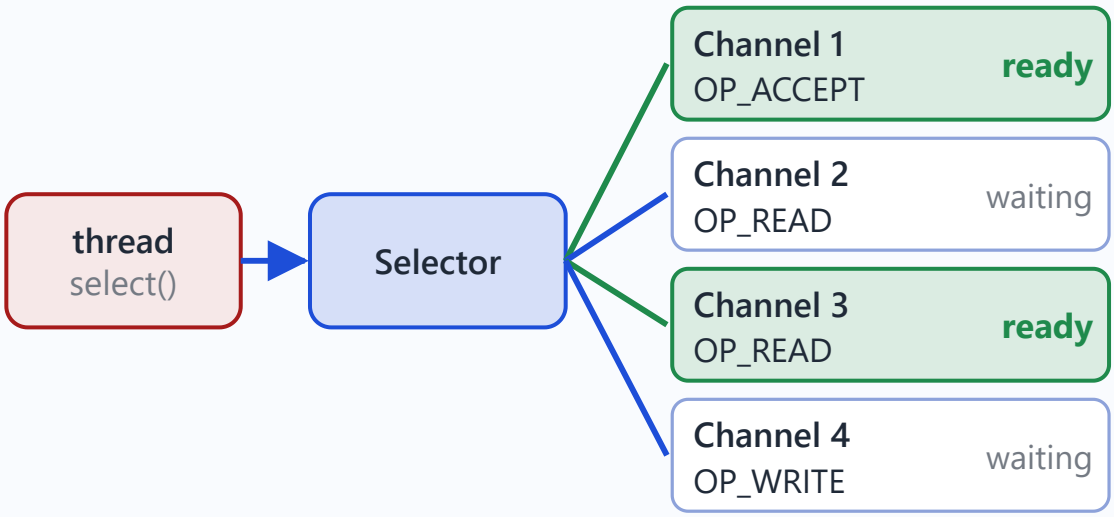


Figure 8.30: A selector. A single thread watches four channels with `select()` and handles only the ready ones.

Constant	Meaning
`OP_READ`	the channel can be read from
`OP_WRITE`	the channel can be written to
`OP_CONNECT`	establishing the connection has finished
`OP_ACCEPT`	a new connection can be accepted

Table 8.25: The `SelectionKey` event constants.

The following skeleton is the selector loop of a server. The loop is infinite; this program was therefore checked only by compiling it.

```
public class SelectorDemo {
  public static void main(String[] args) throws IOException {
    ServerSocketChannel server = ServerSocketChannel.open();
    Selector selector = Selector.open();
    server.bind(new InetSocketAddress(8080));
    server.configureBlocking(false);
    server.register(selector, SelectionKey.OP_ACCEPT);
    while (true) {
      int ready = selector.select();
      if (ready > 0) {
        Set<SelectionKey> keys = selector.selectedKeys();
        for (SelectionKey key : keys) {
          if (key.isAcceptable()) {
            // accept a connection
          } else if (key.isReadable()) {
            // read data
          }
        }
        keys.clear(); // drop handled keys
      }
    }
  }
}
```

The server channel is bound to port 8080, put into non-blocking mode, and registered for `OP_ACCEPT`. In the loop, `select()` gives the number of ready channels and `selectedKeys()` gives the ready keys. If the handled keys are not removed with `clear()`, they come back in the next round. In a real server, the accepted client channel is also registered with `OP_READ` and its data is read. Two rules are checked at registration. A blocking channel cannot be registered, and a server channel supports only the `OP_ACCEPT` event:

```
registering a blocking channel: IllegalBlockingModeException
OP_READ on a server channel: IllegalArgumentException
validOps == OP_ACCEPT: true
```

8.10.9 Pros and cons of NIO

The advantages of NIO fall under three headings. **Performance:** data is transferred in blocks, which is more efficient than classic I/O for large files. **Flexibility:** buffers and channels allow flexible data handling; channels are two-way, and files and the network are processed in the same way. **Many connections:** with a selector, many channels are watched from a single thread, and only the active channels are processed.

The drawback is complexity. The code is longer and harder than with classic I/O. The buffer state and error cases such as a partial read or a dropped connection have to be managed by hand. Learning the selector also takes time, because the state of many channels has to be tracked correctly. As a rule,

`BufferedReader` or `Files` is enough for reading a single file line by line. Channels come into their own for very large files and for large numbers of network connections.

8.11 Solved problems and exercises

Example 8.2

The numbers 12 and 345 are written first as text, then as binary. Which two numbers does the program print?

```
try (PrintWriter pw = new PrintWriter("s.txt")) {
  pw.print(12);
  pw.print(' ');
  pw.print(345);
}
System.out.println(new File("s.txt").length());
try (DataOutputStream d = new DataOutputStream(new FileOutputStream("s.dat"))) {
  d.writeInt(12);
  d.writeInt(345);
}
System.out.println(new File("s.dat").length());
```

`PrintWriter` converts the numbers to characters: `"12 345"`, six characters. Since all the characters are ASCII, the file is 6 bytes. `DataOutputStream` writes two `int` values: $2 \cdot 4 = 8$ bytes. The output is **6** and **8**. Here the text came out shorter because the numbers have few digits.

Example 8.3

`a.txt` does not exist at the start. What does the program print?

```
try (FileWriter w = new FileWriter("a.txt")) {
  w.write("ab");
}
try (FileWriter w = new FileWriter("a.txt")) {
  w.write("cd");
}
try (FileWriter w = new FileWriter("a.txt", true)) {
  w.write("ef");
}
try (BufferedReader r = new BufferedReader(new FileReader("a.txt"))) {
  System.out.println(r.readLine());
}
```

The first `FileWriter` creates the file and writes `ab`. The second `FileWriter` is not in append mode; it resets the file, `ab` is erased, and `cd` is written. The third appends with `true`. The file has a single line, and the output is `cdef`.

Example 8.4

What does the program print on its three lines?

```
ByteBuffer b = ByteBuffer.allocate(10);
b.put((byte) 1).put((byte) 2).put((byte) 3);
b.flip();
b.get();
System.out.println(b.position() + " " + b.limit() + " " + b.remaining());
b.compact();
System.out.println(b.position() + " " + b.limit());
b.clear();
System.out.println(b.position() + " " + b.limit() + " " + b.get(0));
```

After the three `put` calls, the position is 3. `flip` sets the position to 0 and the limit to 3. One `get` advances the position to 1; the remaining count is 3 − 1 = 2. The first line is `1 3 2`. `compact` moves the unread 2 and 3 to the front; the position becomes the remaining count, 2, and the limit the capacity, 10. The second line is `2 10`. `clear` sets the position to 0 and the limit to 10 but does not erase the data; `get(0)` gives the 2 that was moved to the front. The third line is `0 10 2`.

Example 8.5

What does the program print?

```
try (RandomAccessFile f = new RandomAccessFile("r.dat", "rw")) {
  for (int i = 1; i <= 4; i++) f.writeInt(i * 10);
  f.seek(4);
  int a = f.readInt();
  f.seek(f.length() - 4);
  int b = f.readInt();
  System.out.println(a + " " + b + " " + f.getFilePointer());
}
```

The file has four records: 10, 20, 30, 40; each is 4 bytes, 16 bytes in all. `seek(4)` is the start of the second record, so `a` = 20. `length() - 4` = 12 is the start of the last record; `b` = 40. After the read the pointer is at 16, that is, at the end of the file. The output is `20 40 16`.

The solutions of the four exercises in the slides were also verified by running them with JDK 21. The first counts the lines, words, and characters of `text.txt`, whose content is "Java and files / read and write / (blank line) / very easy":

```
int lineCount = 0, wordCount = 0, charCount = 0;
try (BufferedReader br = new BufferedReader(new FileReader("text.txt"))) {
  String line;
  while ((line = br.readLine()) != null) {
    lineCount++;
    charCount += line.length();
    String trimmed = line.trim();
    if (!trimmed.isEmpty()) wordCount += trimmed.split("\\s+").length;
  }
} catch (IOException e) {
  System.err.println("Error: " + e.getMessage());
}
System.out.println(lineCount + " lines, " + wordCount + " words, "
    + charCount + " characters");
```

```
4 lines, 8 words, 37 characters
```

The `trim()` and `isEmpty()` check is needed for the blank line. `"".split("\\s+")` returns an array of length 1; without the check, the blank line would count as one word. The characters are 14 + 14 + 0 + 9 = 37; `readLine` discards the line terminators, so they are not counted.

The second exercise reads `n` from the keyboard, writes the numbers from 1 to `n` and their squares to `squares.txt`, then reads the file and finds the sum of the squares:

```
Scanner keyboard = new Scanner(System.in);
try (PrintWriter pw = new PrintWriter(new FileWriter("squares.txt"))) {
  int n = keyboard.nextInt();
  for (int i = 1; i <= n; i++) {
    pw.println(i + " " + i * i);
  }
}
int total = 0;
try (Scanner file = new Scanner(new File("squares.txt"))) {
  while (file.hasNextInt()) {
    file.nextInt();               // i
    total += file.nextInt();      // i * i
  }
}
System.out.println("Sum of squares: " + total);
```

```
Sum of squares: 55
```

Reading starts after the writing block has closed; otherwise the data would remain in the buffer and the file would be read as empty. For `n` = 5: 1 + 4 + 9 + 16 + 25 = 55. The third exercise serializes an `int` array and reads it back. Arrays are serializable:

```
int[] numbers = {3, 1, 4, 1, 5};
try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("numbers.ser")))
{
  out.writeObject(numbers);
}
try (ObjectInputStream in = new ObjectInputStream(new FileInputStream("numbers.ser"))) {
  int[] loaded = (int[]) in.readObject();
  int total = 0;
  for (int x : loaded) total += x;
  System.out.println(loaded.length + " elements, total " + total);
}
```

```
5 elements, total 14
```

In the fourth exercise, five grades are written as `int` values with `RandomAccessFile`, and the grade at a given sequence number is found without reading the whole file: `f.seek((pos - 1) * 4); int grade = f.readInt();`. The position of the third grade is 8. The same approach works for the last record of a log file with 10,000 lines, but only if the records have a fixed size: `seek((n - 1) * size)`. If the lines differ in length, the position cannot be computed. Either the file is read sequentially from the start, or a `\n` is searched for backwards from the end.

8.12 Common mistakes

The mistakes mentioned here and there in the chapter are collected below, together with the real messages of the compiler or the JVM.

Not handling a checked exception. `new FileReader("a.txt")` can throw `FileNotFoundException`, and `readLine()` can throw `IOException`. The code must be inside `try`-`catch`, or the method must say `throws IOException`:

```
Main.java:5: error: unreported exception FileNotFoundException; must be caught or declared
to be thrown
    BufferedReader r = new BufferedReader(new FileReader("a.txt"));
                                          ^
Main.java:6: error: unreported exception IOException; must be caught or declared to be
thrown
    String s = r.readLine();
                          ^
2 errors
```

Forgetting the import. `Scanner` is in `java.util`, and the I/O classes are in `java.io`. If no `import` is written, the compiler cannot find the class:

```
Main.java:3: error: cannot find symbol
    Scanner sc = new Scanner(System.in);
    ^
  symbol:   class Scanner
  location: class Main
Main.java:3: error: cannot find symbol
    Scanner sc = new Scanner(System.in);
                     ^
  symbol:   class Scanner
  location: class Main
2 errors
```

Each of the other mistakes was shown with its real output in the relevant section: writing a single backslash in a Windows path, not checking for `null` in `finally`, using `nextLine` after `nextInt`, not closing a buffered writer, forgetting `flip()` or `Serializable`, keeping the result of `read()` in a `byte`, and writing `write(chunk)` in a block copy. In most of these the program compiles and runs but gives a wrong result. For this reason, after writing I/O code, always check the size of the copy, the values read, and the actual contents of the file.

8.13 Summary

- A stream is a structure that carries data from a source to a destination in order. Every Java program has three standard streams: `System.in`, `System.out`, and `System.err`. When output is redirected, `System.err` stays on the screen.
- `Scanner` reads the keyboard, files, and strings. `nextInt()` leaves the line terminator in the buffer; a following `nextLine()` returns an empty string.
- Every file is a sequence of bytes. In a text file the bytes are converted to letters with a character encoding; in a binary file they are interpreted as numbers, pixels, or instructions.

- ASCII is 7 bits, and ISO-8859-9 is 8 bits for Turkish; UTF-8 writes a character with 1–4 bytes and has been the default since Java 18. A file must be read with the encoding it was written in.
- 127 is 3 bytes as text and 4 bytes as an `int`. The binary size depends on the type, the text size on the number of digits. Binary data is read back in the order and with the type in which it was written.
- `File` is only a path; it does not read the contents. Most of its methods report errors by returning `false`.
- Byte streams (`InputStream` / `OutputStream`) are for any kind of data, and character streams (`Reader` / `Writer`) are for text. Wrapper streams add capabilities; closing the outermost one closes them all.
- `readLine()` returns `null` at the end of the file, and `read()` returns −1. The result of `read()` is kept in an `int`.
- `FileWriter(name, true)` appends; the form without the parameter resets the file. `PrintWriter` does not throw exceptions; errors are seen with `checkError()`.
- Buffering batches disk accesses. Data written may wait in memory until `flush()` or `close()`.
- try-with-resources closes resources in the reverse of the opening order and before `catch`.
- In a block copy, only the `n` bytes actually read are written: `write(chunk, 0, n)`. The shortcuts are `transferTo` and `Files.copy`.
- `RandomAccessFile` jumps to any position with `seek`; with fixed-size records, the position of record k is $k \cdot$ record size.
- Serialization converts an object to bytes. The class implements `Serializable`; `static` and `transient` fields are not written; the constructor is not called on reading back.
- NIO works with buffers, channels, and selectors. After writing to a buffer, call `flip()` before reading and `clear()` or `compact()` before writing again.

8.14 Key terms

stream standard input / output text file / binary file character encoding

byte stream / character stream wrapper stream (decorator) buffer / buffering try-with-resources

checked exception random access file serialization channel selector locale thread

8.15 Test Yourself

1. How many bytes does the word `"Ilık"` take in UTF-8 and in ISO-8859-9?

   Answer

   In UTF-8, `I`, `l`, and `k` take one byte each and `ı` takes two: **5**. In ISO-8859-9 every letter is one byte: **4**.

2. With a `PrintWriter`, `print(5L)`, `print('A')`, and `print(1.5f)` are called; with a `DataOutputStream`, `writeLong(5)`, `writeChar('A')`, and `writeFloat(1.5f)`. How many bytes is each of the two files?

   Answer

   The text file contains `5A1.5`: **5** bytes. In the binary file, `long` is 8, `char` 2, and `float` 4 bytes: **14** bytes. `print(1.5f)` is independent of the locale and writes a dot.

3. A file of 10,000 bytes is read by calling `read(array)` with a 4096-byte array. What values do the calls return? If `write(array)` had been used in the copy, how many bytes would the copy have?

   Answer

   4096, 4096, 1808, and −1 at the end of the file. `write(array)` writes the whole array in every round: 3 · 4096 = **12,288** bytes. In the last round, 2288 stale bytes are written in addition.

4. `ByteBuffer t = ByteBuffer.allocate(5);` then 1, 2, 3 are written with `put`; then `flip()`, two `get()` calls, `compact()`, `put((byte) 9)`, and `flip()` again. What are the position, the limit, the remaining count, and cells 0 and 1?

   Answer

   After the two `get` calls, the only unread element is 3. `compact` moves it to the front: position 1, limit 5. `put(9)` writes to cell 1 and the position becomes 2. `flip` sets the limit to 2 and the position to 0. Result: position **0**, limit **2**, remaining **2**, cells **3** and **9**.

5. Into an empty file, 7, 8, 9 are written as `int` values with `RandomAccessFile`. Then `seek(4); writeInt(100); seek(length()); writeShort(1);` is done. What is the length of the file, and

what does `readShort()` return after `seek(0)`?

> **Answer**
>
> Three `int` values are 12 bytes; `writeInt(100)` is written in place of the 8, and the length does not change. `writeShort` appends 2 bytes at the end: **14**. `readShort()` reads the first two bytes. Since the bytes of 7 are `00 00 00 07`, the result is **0**.

6. An object is created of a serializable class whose constructor prints "constructed" and which has the fields `transient String note = "temporary";` and `int n = 5;`; `n` is set to 8, and the object is written and read back. How many times is "constructed" printed, and what are `note` and `n` in the object read?

> **Answer**
>
> **Once**; `readObject` does not call the constructor. `note` is `null`, because a `transient` field is not written and its field initializer does not run either. `n` is **8**.

7. The block `try (R a = new R("A"); R b = new R("B"); R c = new R("C")) { … }` finishes without an exception; `R` prints a message when it is opened and when it is closed. What is the order of the messages?

> **Answer**
>
> **A opened, B opened, C opened**, the body, **C closed, B closed, A closed**. Resources are opened from left to right and closed in the reverse order.

8. A 3 MB file is read byte by byte through a default `BufferedInputStream`. How many reads are made on the underlying `FileInputStream`?

> **Answer**
>
> 3 · 1,048,576 / 8192 = 384 reads that carry data, plus one read for the end of the file: **385**.

8.16 Further reading

- Oracle, *The Java Tutorials*, "Basic I/O": the official, free source that explains streams, `Scanner` and formatting, and file operations with NIO.2 through short examples.
- C. S. Horstmann, *Core Java, Volume II: Advanced Features* (12th ed.), Chapter 2 "Input and Output": treats stream classes, character encodings, serialization, and memory-mapped files in detail.
- Oracle, *Java Object Serialization Specification*: defines the format of serialized bytes, how `serialVersionUID` is computed, and the rules of version compatibility.

- OpenJDK, JEP 400 "UTF-8 by Default": explains why the default encoding was made UTF-8 in Java 18 and what the effect on older programs is.

8.17 References

1. Oracle, *The Java Tutorials*: "Basic I/O" (I/O Streams, File I/O featuring NIO.2), "Scanning and Formatting".
2. Oracle, *Java SE 21 API Documentation*: the packages `java.io`, `java.nio`, `java.nio.channels`, and `java.nio.file`.
3. Oracle, *Java Object Serialization Specification*.
4. J. Gosling et al., *The Java Language Specification, Java SE 21 Edition* (3.10.6 escape sequences, 14.20.3 try-with-resources).
5. P. Deitel and H. Deitel, *Java: How to Program, Early Objects*, 11th ed., Pearson, 2017 (Chapter 15: Files, Input/Output Streams, NIO and XML Serialization).
6. C. S. Horstmann, *Core Java, Volume II: Advanced Features*, 12th ed., 2022 (Chapter 2: Input and Output).
7. K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly, 2022 (Chapter 16: Saving Objects (and Text)).
8. A. B. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019.
9. OpenJDK, *JEP 400: UTF-8 by Default*, JDK 18, 2022. https://openjdk.org/jeps/400

PART IV · CHAPTER 9

Graphics and User Interfaces

Until now our programs have run in the console: the program asked a question, the user typed an answer, and the results were printed one under another. The programs people use every day work with windows, buttons, text fields, and drawings. This chapter shows how to build a graphical user interface in Java. It begins with how an event-driven program differs from a console program, then covers the AWT and Swing libraries and screen coordinates. Next come windows built with `JFrame` and `JPanel`, drawing with `paintComponent` and `Graphics`, and the `Point`, `Color`, and `Font` classes. Layout managers arrange components in the window, and we trace pixel by pixel how a layout is computed. Mouse events, listener interfaces, adapter classes, and the event dispatch thread are the heart of the chapter. It ends with three small applications: a mouse drawing program, a text converter, and a change calculator.

LEARNING OUTCOMES

Build a window
Create a window and a panel with `JFrame` and `JPanel`, add components, show the window, and close it.

Draw
Override `paintComponent` and draw shapes and text with `Graphics`; use screen coordinates and bounding rectangles.

Lay out components
Arrange components with `BorderLayout`, `FlowLayout`, `GridLayout`, and `BoxLayout`; compute the position and size of each component.

Respond to events
Handle mouse, button, and text events with listener interfaces, adapter classes, anonymous classes, and lambda expressions.

Start safely
Explain why the interface is built on the event dispatch thread with `SwingUtilities.invokeLater`; trace the difference between `invokeLater` and `invokeAndWait`.

Write applications
Read and modify the drawing program, the text converter, and the change calculator.

9.1 Graphics programming

9.1.1 Components, events, and painting

Most people use a computer through windows. A window has a title, buttons, menus, and text fields. The user clicks with the mouse, types on the keyboard, and resizes the window, and the program responds to these actions. An interface of this kind is called a graphical user interface.

Definition 9.1

A **graphical user interface** (GUI) is an interface in which the user interacts with a program through visual elements such as windows, buttons, menus, text fields, and drawings.

Java provides an extensive class library for building GUIs. To write a GUI program with it, you need to understand three concepts.

- **Component**: The building blocks of the interface, such as windows, buttons, labels, and text fields. Every component is an object of some class. A button is an object created from the `JButton` class; its text, color, and size are stored in the fields of that object.
- **Event handling**: User actions such as a mouse click or a key press are captured, and methods that respond to them are called.
- **Painting**: Shapes are drawn, the screen is updated, and components are displayed.

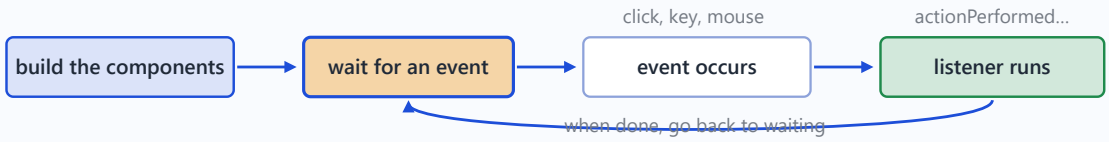


Figure 9.1: The life of a GUI program. The components are built, the program waits for an event, and when an event arrives the matching listener method runs; then the program waits again.

A GUI program is like a shop waiting for customers. First the window display is prepared: the components are created and placed in the window. Then the program waits. Whatever the customer touches, the assistant responsible for that item springs into action. In the program, the display corresponds to the components, the touch to an event, and the assistant to a listener method (Figure 9.1).

9.1.2 Console programs and GUI programs

In a console program the flow is sequential. The `main` method runs from beginning to end. When the program needs input, it asks the question itself and reads a line from the keyboard with `Scanner`. The result is printed with `System.out.println`. When `main` ends, the program ends.

A GUI program flows differently. `main` only builds the interface and returns. After that the program waits for the user's move. When the user presses a button or clicks a panel, the matching method runs. This style of execution is called event-driven programming.

Definition 9.2

An **event** is a user action, such as a click, a key press, or a mouse drag, or a notification from the system. In an **event-driven** program, the order in which events arrive, not the order of lines in the code, determines the flow.

	Console program	GUI program
Flow	Sequential: `main` runs from start to end, and the program asks the questions	Event-driven: the program waits for the user's move
Input	A line is read from the keyboard with `Scanner`	Buttons, text fields, and the mouse; listener methods are called
Output	`System.out.println`	The text of components changes, or the panel is repainted
Termination	The program ends when `main` ends	Keeps running until the window is closed, even after `main` ends
Library	`java.util`, `java.io`	`java.awt`, `javax.swing`, `java.awt.event`

Table 9.1: Comparison of console programs and GUI programs.

The last two rows of the table matter. In a GUI program, the program keeps running even though `main` has finished. The reason is that Java starts a separate thread when a window opens. This thread waits on the event queue and passes each incoming event to the matching listener. The JVM does not exit while this thread is running.

In a GUI program, the answer to “what happens next?” is not the order of the code; it is the user's next action. That is why most of the code consists of listeners of the form “when this happens, do that.” The programmer writes a response for each action but does not know in what order the actions will arrive.

Common mistake

Trying to wait for user input in a `while` loop in a GUI program. A “loop until the user presses the button” written out of console habit keeps the event thread busy, and the window freezes. In a GUI program Java does the waiting; the programmer only registers the listener.

9.1.3 AWT and Swing

Java's graphics classes are spread over two libraries. Both come with the JDK; nothing extra needs to be installed.

- **AWT** (Abstract Window Toolkit), package `java.awt`: Java's first graphics library. Today it is used mostly for its shared classes: `Graphics`, `Color`, `Font`, `Point`, the layout managers, and the event classes (`java.awt.event`).
- **Swing**, package `javax.swing`: a library built on top of AWT in which Java itself draws the components. Class names start with `J`: `JFrame`, `JPanel`, `JButton`, `JLabel`.

The difference between the two libraries lies in how components are drawn. When AWT needs a button, it uses the operating system's own button, so the same program looks different on Windows and on Linux. Swing paints the button itself, so the look is the same on every system and is under Java's control. In this chapter the components come from Swing, while the drawing and helper classes come from AWT. A typical GUI program therefore imports both packages: `import java.awt.*;` and `import javax.swing.*;`.

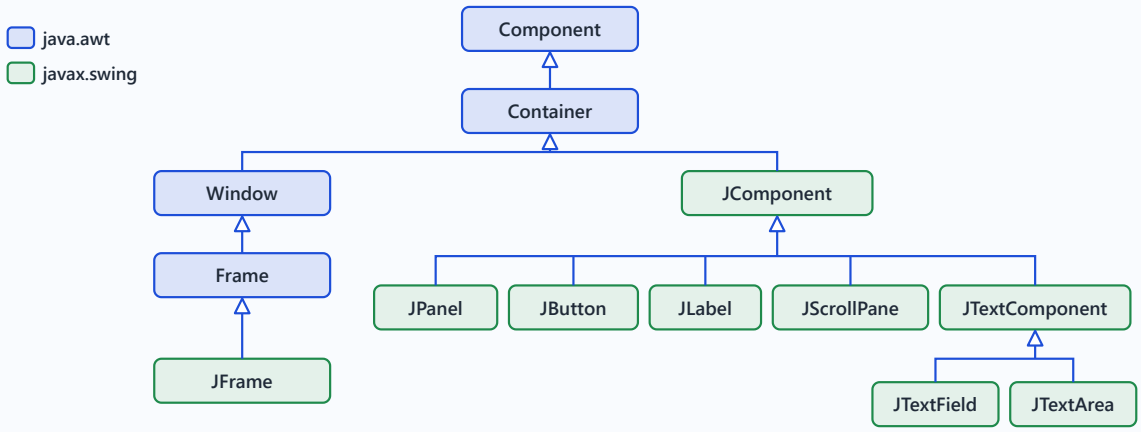


Figure 9.2: The inheritance tree of the GUI classes. A hollow triangle means "is derived from." `JFrame` is an AWT window; the other Swing components derive from `JComponent`.

Figure 9.2 shows how inheritance is used in the GUI library. At the root of the tree is `Component`: anything with a position and size on the screen is a component. A `Container` is a component that can hold other components. Two branches leave `Container`. The first holds `Window`, `Frame`, and finally `JFrame`. The second holds `JComponent`, the common superclass of the Swing components. `JPanel`, `JLabel`, and the text components derive from `JComponent` directly or by a short path. `JButton` has the `AbstractButton` class in between.

This tree has a practical consequence. If a method expects a `Component` as a parameter, you can pass it a button, a label, or a panel. Polymorphism is an everyday tool here. For example, the `panel.add(...)` method accepts any component.

9.1.4 Screen coordinates

A graphical application is built in three steps. First, components are added to the interface using Java's graphics classes. Second, event-handling methods are defined that respond to the user's actions. Finally, the visuals are produced with methods such as `paint()` or `paintComponent()`. Before we start drawing, we need to know how the screen is addressed.

The screen is a grid of pixels. Every pixel has a column number (`x`) and a row number (`y`). Coordinates are integers measured from the top-left corner of the component.

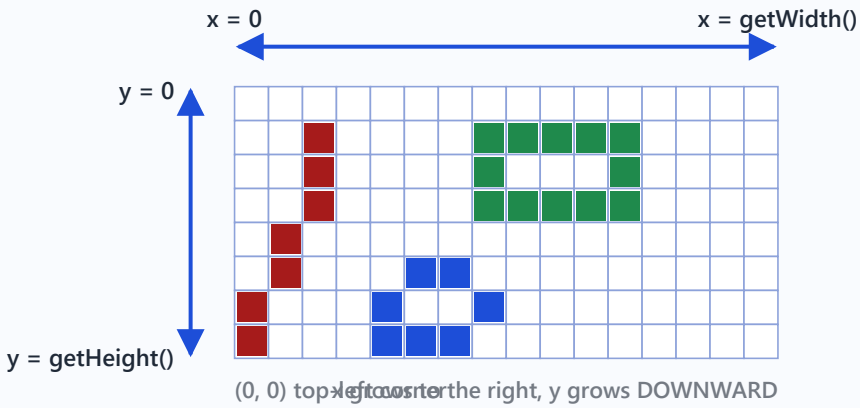


Figure 9.3: The screen coordinate system. The origin is at the top-left corner; `x` grows to the right and `y` downward. The line, rectangle, and ellipse are made of pixels.

Definition 9.3

Screen coordinates identify a pixel inside a component by an integer pair `(x, y)`. `(0, 0)` is the top-left pixel of the component. `x` grows to the right and `y` downward. The bottom-right pixel is `(getWidth() - 1, getHeight() - 1)`.

The most important difference from the axes of mathematics is the direction of `y`. In mathematics `y` grows upward; on the screen it grows downward. This choice comes from the history of displays: cathode-ray tubes drew the picture from top to bottom, line by line. As a result, a point whose `y` value increases moves down the screen. Decreasing `y` moves the point up.

Example 9.1

A panel is 300 pixels wide and 160 pixels high. What are the coordinates of the top-right, bottom-left, and center pixels?

Columns are numbered 0 to 299 and rows 0 to 159. The top-right pixel is `(299, 0)` and the bottom-left pixel is `(0, 159)`. For the center pixel, the width and height are divided by two: `(300 / 2, 160 / 2) = (150, 80)`. If the numbers were odd, integer division would discard

the fraction; for example, 261 / 2 = 130.

9.1.5 Canvas, pen, color, and font

Drawing needs three things: a surface to draw on, a pen that does the drawing, and the pen's properties.

- **Canvas**: a surface on which a picture can be drawn. In Swing, a `JPanel` is usually used. In AWT there is a `Canvas` class for this job. The programmer derives a subclass from `JPanel` and draws inside it.
- **The `Graphics` class**: the "pen" that does the drawing. Its methods have names such as `drawLine()`, `drawRect()`, `fillOval()`, `drawString()`, and `drawImage()`.
- **`Color` and `Font`**: `Color` sets the pen's color and `Font` sets its typeface: `g.setColor(Color.RED)`, `g.setFont(new Font("Serif", Font.BOLD, 24))`.

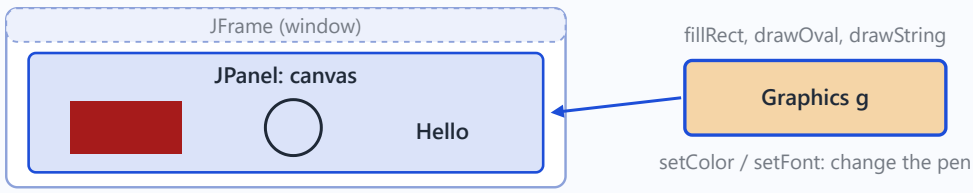


Figure 9.4: A `JPanel` canvas inside a `JFrame` window. The `Graphics` object does the drawing.

The pen analogy also explains a rule. The `Graphics` object is the pen in the programmer's hand. `setColor` changes the pen. Until the pen is changed again, all drawing uses the same color. The same holds for the font: after a call to `setFont`, every `drawString` uses the new font. A `Graphics` object is therefore a stateful object: it remembers its own color and font.

9.1.6 Drawing with `paintComponent`

The way to draw permanently on a panel is to override the `paintComponent` method in a class derived from `JPanel`.

```
import java.awt.*;
import javax.swing.*;

class DrawPanel extends JPanel {
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);     // clear background
        g.setColor(Color.BLUE);
        g.drawRect(20, 30, 50, 70);  // x, y, w, h
        g.drawString("Hello, Java Graphics!", 100, 100);
    }
}
```

The class extends `JPanel` and overrides the superclass's `paintComponent(Graphics g)` method. The `@Override` annotation tells the compiler that "this method replaces a method of the superclass." If the method name is misspelled, the compiler sees the annotation and reports an error. The first line of the body is `super.paintComponent(g)`: it performs the superclass's own painting, which fills the background with the background color. Then the pen is switched to blue. `drawRect(20, 30, 50, 70)` draws the outline of a rectangle whose top-left corner is at `(20, 30)`, 50 pixels wide and 70 pixels high. The last line writes a text at the point `(100, 100)`.

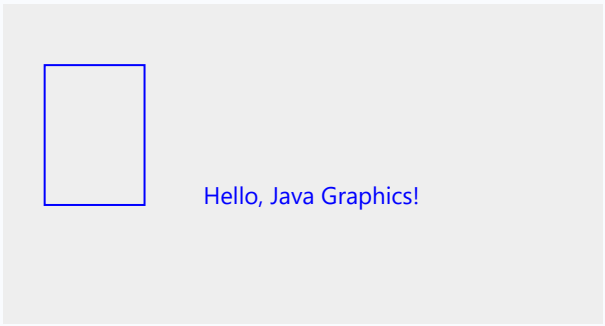


Figure 9.5: The output of `DrawPanel` for a 300 × 160 panel: a blue outline and text in the same color.

Notice something in this code: nothing in the program ever calls `paintComponent`. Swing calls it. Swing runs the method when the window first opens, when its size changes, when another window that covered it is dragged away, or when the program requests a repaint with `repaint()`. When it calls the method, it passes a ready-made `Graphics g` object, which is tied to the region currently being painted.

To see which drawing calls the method actually makes, we can use Swing's `DebugGraphics` class. It is a pen that writes each drawing call as a line of text. Running `DrawPanel` with this pen on a 300 × 160 image shows the following calls (abbreviated):

```
setColor(238, 238, 238)
fillRect(0, 0, 300, 160)
setColor(0, 0, 255)
drawRect(20, 30, 50, 70)
drawString("Hello, Java Graphics!", 100, 100)
```

The programmer did not write the first two lines. They are the work of `super.paintComponent(g)`: the pen is switched to the panel's background color, the light gray `(238, 238, 238)`, and the whole panel is filled with it. The next three lines are the calls the programmer wrote.

Definition 9.4

`paintComponent(Graphics g)` is the method with which a Swing component draws its own contents. Swing calls this method whenever the component needs to be repainted. A class that wants to do its own drawing overrides the method and writes `super.paintComponent(g)` first in the body.

A graphics object can also be obtained with `panel.getGraphics()`. However, drawing done with this object is not permanent. The next time the window is repainted, for example after it is minimized and restored, Swing calls `paintComponent` and whatever was drawn through `getGraphics()` is erased. The way to draw permanently is `paintComponent`. The data of the drawing is kept in fields, and the method regenerates the picture from this data on every call.

The position of text also has a subtlety. The `y` parameter of `drawString` is not the top edge of the text but its baseline. The baseline is the line on which the letters "sit." In the call `drawString("Hello, Java Graphics!", 100, 100)`, the bodies of the letters stay above row 100; only the tails of letters such as "g" and "j" drop below that line.

Common mistake

Writing `drawString(text, x, 0)` and expecting the text to appear at the very top of the panel. Because `y = 0` is the baseline, almost all of the letters end up outside the panel, in the invisible region. For the text to be visible, `y` must be at least the height of the font; for a 24-point font, a value around 30–40 is safe.

9.1.7 `Graphics` drawing methods

The commonly used methods of the `Graphics` class are collected in Table 9.2. The names follow a pattern: methods starting with `draw…` draw only the outline, and those starting with `fill…` draw the shape filled.

Method	What it does
`drawLine(x1, y1, x2, y2)`	A line between two points
`drawRect(x, y, w, h)`	A rectangle outline; `(x, y)` is the top-left corner
`fillRect(x, y, w, h)`	A filled rectangle
`drawOval(x, y, w, h)`	The ellipse that fits in this rectangle (a circle if w = h)
`fillOval(x, y, w, h)`	A filled ellipse or disk
`drawString(s, x, y)`	Text; `y` is the baseline
`setColor(c)` / `setFont(f)`	The color or font of later drawing
`drawImage(image, x, y, null)`	Draws an image at the point `(x, y)`

Table 9.2: The basic drawing methods of the `Graphics` class.

Ellipses and circles are specified not by their centers but by their bounding rectangles: you give the top-left corner and the size of the rectangle in which the ellipse fits.

Definition 9.5

The **bounding rectangle** of a shape is the smallest axis-parallel rectangle that completely encloses it. `drawOval(x, y, w, h)` draws the ellipse inside the rectangle whose top-left corner is `(x, y)` and whose size is `w × h`.

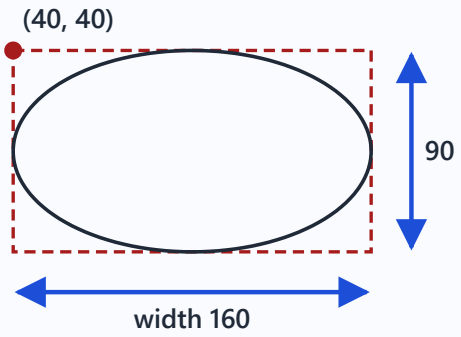


Figure 9.6: `drawOval(40, 40, 160, 90)`. The ellipse fits in the 160 × 90 rectangle whose top-left corner is `(40, 40)`.

To draw a circle with center `(cx, cy)` and radius `r`, compute the top-left corner of the bounding square. The square extends `r` to the left of and above the center, and its side is `2r`. The result is `fillOval(cx - r, cy - r, 2r, 2r)`. Conversely, for a given call `drawOval(x, y, w, h)`, the center of the ellipse is `(x + w / 2, y + h / 2)`.

Example 9.2

Where is the center of the ellipse drawn by `g.drawOval(50, 50, 100, 60)`? Which range of pixels does it cover?

The center is `(50 + 100 / 2, 50 + 60 / 2) = (100, 80)`. The ellipse starts at 50 horizontally and at 50 vertically. If it is drawn on an image and the painted pixels are counted, `x` runs from 50 to 150 and `y` from 50 to 110. Methods that draw outlines also paint the right and bottom edge of the boundary: a `draw…` shape is `w + 1` pixels wide and `h + 1` pixels high, whereas `fill…` paints exactly `w × h` pixels.

The difference in the last sentence is small but real. In the call `drawRect(20, 30, 50, 40)`, the bottom-right corner pixel of the outline is `(70, 70)`, and `(71, 70)` stays empty. With the same arguments, `fillRect` paints up to `(69, 69)`. When an outline is drawn over a filled rectangle, this one-pixel difference becomes visible.

9.1.8 Code trace: the drawing order of shapes

The following panel uses each of the basic methods once.

```
class ShapesPanel extends JPanel {
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawLine(10, 10, 110, 60);
        g.drawRect(130, 10, 80, 50);
        g.fillRect(230, 10, 80, 50);
        g.drawOval(10, 80, 100, 60);
        g.fillOval(130, 80, 60, 60);
        g.drawString("Hello", 230, 115);
    }
}
```

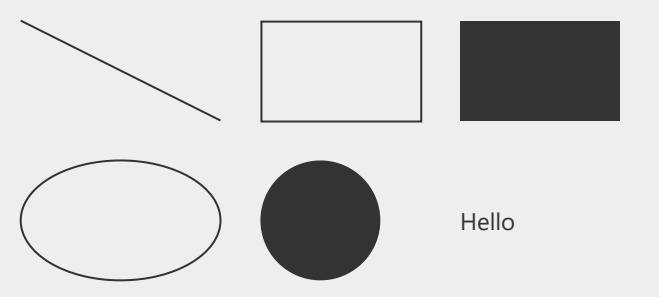


Figure 9.7: The output of `ShapesPanel`: a line, a rectangle outline, a filled rectangle, an ellipse, a filled disk, and text.

There is no `setColor` call anywhere in the code. When no color is given, the pen draws with the panel's foreground color. In Swing's default look this is the dark gray `(51, 51, 51)`; in a running program `getForeground()` returns this value. Table 9.3 traces the steps of the method.

Step	Line	Variables	Explanation
1	4: `super.paintComponent(g);`	g=Graphics (dark gray pen)	Swing calls this method itself. `super.paintComponent(g)` **clears** the canvas with the background color.
2	5: `g.drawLine(10, 10, 110, 60);`	x1=10, y1=10, x2=110, y2=60	A line between two points. Because y grows downward, the line goes to the right and **down**.
3	6: `g.drawRect(130, 10, 80, 50);`	x=130, y=10, w=80, h=50	`drawRect` : (x, y) is the **top-left corner**; only the outline is drawn, and the inside stays empty.
4	7: `g.fillRect(230, 10, 80, 50);`	x=230, y=10, w=80, h=50	`fillRect` : a rectangle of the same size, filled with the pen's color.
5	8: `g.drawOval(10, 80, 100, 60);`	x=10, y=80, w=100, h=60	`drawOval` : an ellipse that **fits** in the dashed box. (10, 80) is the corner of the box, not of the ellipse.
6	9: `g.fillOval(130, 80, 60, 60);`	x=130, y=80, w=60, h=60	Since w = h, a filled **disk**: center (160, 110), radius 30.
7	10: `g.drawString("Hello", 230, 115);`	s="Hello", x=230, y=115	In `drawString` , y is the **baseline** of the text: the letters are written above 115.
8	11: `}`	—	The method ends; every time the window is repainted (a size change, `repaint()`), everything is drawn again **from the start** in the same order.

Table 9.3: Trace of `ShapesPanel.paintComponent` .

The line in step 2 goes from `(10, 10)` to `(110, 60)` . Because the second point has a larger `y` , the line goes right and down. Steps 3 and 4 draw two rectangles of the same size. One is only an outline; the other is filled with the pen's color. In step 5, the point `(10, 80)` is the corner of the bounding box, not of the ellipse.

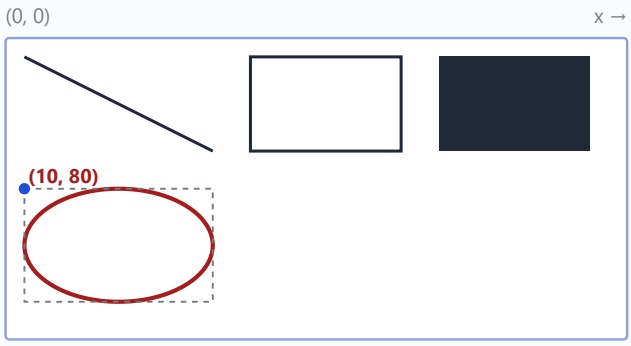


Figure 9.8: Step 5. `drawOval(10, 80, 100, 60)` : the ellipse fits in the bounding box shown with a dashed line.

In step 6, the width and height are equal, so a filled disk is drawn. The center is `(130 + 30, 80 + 30) = (160, 110)` and the radius is 30. In step 7, the baseline of the text is `y = 115`; the letters are written above this line. When the method ends, the panel is complete. Every time the window is repainted, the same calls are made again from the start in the same order.

Drawings are painted on top of one another. A shape drawn later covers an earlier one. In this program the shapes do not overlap, so the order is not visible. When shapes overlap, the one drawn last stays on top.

Common mistake

Forgetting the line `super.paintComponent(g)`. This line clears the background. Without it the panel is not painted with its background, and leftover images from the previous drawing or from other windows may remain on the screen. In animations in particular, a moving shape leaves a trail wherever it has been.

9.2 Windows and panels

9.2.1 The `JFrame` class

All the components of a GUI program live inside a window. In Swing, a window is created with the `JFrame` class.

Definition 9.6

`JFrame` is a top-level window with a title bar and minimize and close buttons on the screen. It is not placed inside another component; other components are added to its content pane.

Code	Purpose
`JFrame frame = new JFrame("Title");`	Creates a window with the given title (not yet visible)
`frame.add(component);`	Adds a component to the window (to the content pane)
`frame.setSize(400, 300);`	Width and height (pixels, including the borders)
`frame.pack();`	Sets the size to the size preferred by the components inside
`frame.setLocation(x, y);`	The position of the window on the screen; `setLocationRelativeTo(null)` centers it
`frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`	Ends the program when the window is closed
`frame.setVisible(true);`	Makes the window visible (usually called last)

Table 9.4: Frequently used calls with `JFrame`.

There are two ways to size a window. `setSize` gives the size directly; this size includes the title bar and the borders. `pack()` adds up the sizes preferred by the components inside and makes the window exactly that large. In a window whose components are arranged by a layout manager, `pack()` usually gives a better result, because the content fits in the window even when the font or the screen resolution changes.

The `add` call actually adds the component not to the window itself but to its **content pane**. The content pane is the inner area below the title bar. In old Java versions you had to write `frame.getContentPane().add(...)`. Since Java 5, `frame.add(...)` makes this call itself. The examples in this chapter use both forms; they mean the same thing.

9.2.2 The first window

```
import javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class JFrameDemo {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            JFrame frame = new JFrame("Java JFrame Example");
            frame.setSize(400, 300);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
        });
    }
}
```

The program imports the `JFrame` and `SwingUtilities` classes. `main` does not build the window itself; it leaves the building to the event dispatch thread (EDT) with `SwingUtilities.invokeLater`. Swing components should be used only from this thread; the reason is explained in 9.6.5. The code running on the EDT creates a window titled “Java JFrame Example,” sets its size to 400 × 300, sets the close behavior, and makes the window visible. The window is empty. `main` ends as soon as it has queued the work, but the program keeps running until the window is closed.

This short program has three pitfalls:

- If `setVisible(true)` is forgotten, the window is created in memory but never appears on the screen. The program raises no error; it simply looks as if nothing is happening.
- If `EXIT_ON_CLOSE` is not given, the default behavior is `HIDE_ON_CLOSE`. When the user presses the × button, the window is hidden, but the program keeps running in the background. In a development environment this shows up as a pile of programs that were forgotten and never stopped.
- If no size is given, the window opens at nearly 0 × 0, just the size of its title bar.

The four constants accepted by `setDefaultCloseOperation` are defined in the `WindowConstants` interface. `DO_NOTHING_ON_CLOSE` (0) prevents closing altogether. `HIDE_ON_CLOSE` (1) hides the window. `DISPOSE_ON_CLOSE` (2) destroys the window; if no other windows remain, the program may end. `EXIT_ON_CLOSE` (3) calls `System.exit(0)`.

If this program is run in an environment with no display, it produces an interesting result. Java recognizes a display-less environment as *headless* mode. In this mode, the call `new JFrame(...)` throws `java.awt.HeadlessException`, because there is no screen to show it on. Components such as panels, buttons, and text fields can still be created in this mode. Most of the outputs given in this chapter were obtained this way, by running the components directly without opening a window.

9.2.3 The `JPanel` class

Definition 9.7

A **container** is a component that holds other components. `JPanel` is the general-purpose container in the `javax.swing` package; it is used to group and arrange the components inside a window and to draw on.

`JPanel` has four uses:

- **Grouping:** Complex layouts are built by nesting several panels. For example, the row of buttons at the bottom of a window is collected in a separate panel.
- **Customizing:** `setBackground()`, `setBorder()`, and `setPreferredSize()` change its color, its border, and its preferred size.
- **Serving as a canvas:** Drawings, animations, and graphics are done on a panel: derive from `JPanel` and override `paintComponent`.
- **Splitting up code:** Each panel can be a class of its own. A large interface is thus divided into small pieces that can be understood separately.

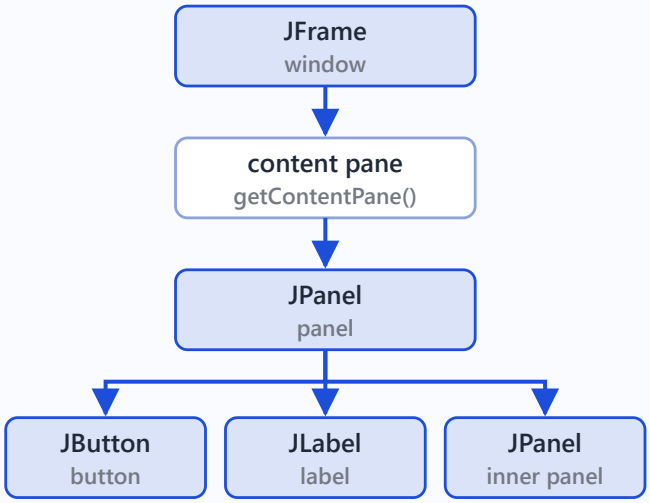


Figure 9.9: Nested components. A `JFrame` contains the content pane, the content pane contains a `JPanel`, and this panel contains a button, a label, and another panel.

The structure in Figure 9.9 is a tree. The root is the window and the leaves are components such as buttons and labels. The interior nodes are containers. When a container is repainted, it first paints itself and then the components inside it. The same rule applies at every level of the component tree.

9.2.4 A panel example

```
import java.awt.Color;
import javax.swing.*;

public class PanelDemo {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            JPanel panel = new JPanel();
            panel.setBackground(Color.LIGHT_GRAY);
            panel.add(new JButton("Click"));
            JFrame frame = new JFrame("JPanel Example");
            frame.add(panel);
            frame.setSize(300, 150);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
        });
    }
}
```

Like the previous example, the program builds the interface on the event dispatch thread with `invokeLater`. It creates a panel and sets its background to `Color.LIGHT_GRAY`, that is, the light gray `(192, 192, 192)`. A button labeled “Click” is added to the panel. Then the window is created, the

panel is added to it, and the window is shown at 300 × 150. The button has no listener; when clicked it looks pressed, but nothing happens.

In the window, the button sits at the top center of the panel, and the panel fills the whole window. Layout managers explain both observations. The default layout manager of `JPanel` is `FlowLayout`. This manager arranges components at their preferred sizes, row by row, centered. Since the panel holds a single button, the button is placed in the middle of the first row. The content pane of a `JFrame` uses `BorderLayout`. A panel added without a region goes to the center region, and the center takes all the remaining space.

Example 9.3

Let this panel be 284 × 111 pixels. If the preferred size of the "Click" button is 61 × 26, what is the position of the button?

By default, `FlowLayout` leaves a gap of 5 pixels between components and at the edges. The available width is 284 − 2 × 5 = 274. The button is placed in the middle of this width: `x = 5 + (274 - 61) / 2 = 5 + 106 = 111`. The row starts 5 pixels below the top edge: `y = 5`. When the panel is brought to this size in headless mode and the layout is computed, the bounds of the button come out as `[x=111, y=5, width=61, height=26]`. The size of the button depends on the font; on another system it may differ by a few pixels.

9.3 Drawing tools

Three small classes are used constantly in drawing: `Point`, which holds a point; `Color`, which holds a color; and `Font`, which holds a typeface. All three are in the `java.awt` package.

9.3.1 The `Point` class

Definition 9.8

`java.awt.Point` is the class that holds a point `(x, y)` on the screen. Its fields `x` and `y` are `public int`. `getX()` and `getY()` return the same values as `double`.

`Point` is a deliberate exception to the principle of encapsulation: its fields are `public` and can be read and written directly. Because point coordinates are accessed so often in graphics code, the library chose simplicity. The class also has methods. The following code uses most of them.

```
Point p = new Point(30, 40);
double px = p.getX();
double py = p.getY();
System.out.println(px + " " + py);
System.out.println(p.x + " " + p.y);
System.out.println(p.distance(0, 0));
p.setLocation(60, 80);
System.out.println(p);
p.translate(5, -10);   // 5 right, 10 up
System.out.println(p);
System.out.println(Point.distance(0, 0, 3, 4));
```

```
30.0 40.0
30 40
50.0
java.awt.Point[x=60,y=80]
java.awt.Point[x=65,y=70]
5.0
```

Step	Line	Variables	New output
1	1: `Point p = new Point(30, 40);`	p=(30, 40)	—
2	2: `double px = p.getX();`	p=(30, 40), px=30.0	—
3	3: `double py = p.getY();`	p=(30, 40), px=30.0, py=40.0	—
4	4: `System.out.println(px + " " + py);`	p=(30, 40), px=30.0, py=40.0	`30.0 40.0`
5	5: `System.out.println(p.x + " " + p.y);`	p=(30, 40), px=30.0, py=40.0	`30 40`
6	6: `System.out.println(p.distance(0, 0));`	p=(30, 40), px=30.0, py=40.0	`50.0`
7	7: `p.setLocation(60, 80);`	p=(60, 80), px=30.0, py=40.0	—
8	8: `System.out.println(p);`	p=(60, 80), px=30.0, py=40.0	`java.awt.Point[x=60,y=80]`
9	9: `p.translate(5, -10); // 5 right, 10 up`	p=(65, 70), px=30.0, py=40.0	—
10	10: `System.out.println(p);`	p=(65, 70), px=30.0, py=40.0	`java.awt.Point[x=65,y=70]`
11	11: `System.out.println(Point.distance(0, 0, 3, 4));`	p=(65, 70), px=30.0, py=40.0	`5.0`

Table 9.5: Trace of the `Point` example.

In step 1 the point `(30, 40)` is created. In steps 2 and 3, the values of `getX()` and `getY()` are stored in `double` variables. Step 4 prints these values: `30.0 40.0`. The decimal point shows that the value is a `double`. Step 5 accesses the fields directly; since `p.x` is an `int`, it prints `30 40`. The same number written in two different forms shows how the type affects the output.

In step 6, `distance(0, 0)` computes the distance of the point from the origin. By the Pythagorean theorem, `√(30² + 40²) = √2500 = 50.0`. In step 7, `setLocation(60, 80)` moves the point to a new position. `px` and `py` do not change: they are copies taken in steps 2 and 3 and are no longer connected to the object. In step 8, `println(p)` calls the object's `toString()` method and prints the class name and the fields: `java.awt.Point[x=60,y=80]`.

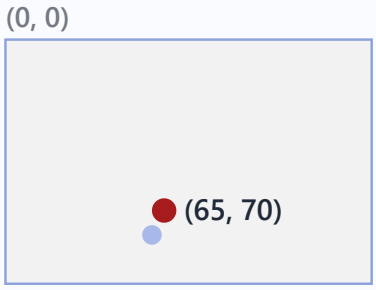


Figure 9.10: Step 9. `translate(5, -10)` moves the point 5 pixels to the right and 10 pixels up: from `(60, 80)` to `(65, 70)`.

`translate(dx, dy)` in step 9 is a relative shift: `dx` is added to `x` and `dy` to `y`. Since `dy = −10`, `y` decreases from 80 to 70 and the point moves up the screen (Figure 9.10). `Point.distance(0, 0, 3, 4)` in the last line is a static method. It needs no object; it takes the coordinates of two points and returns `5.0`.

Example 9.4

After the lines `Point p = new Point(10, 20); Point q = p; q.translate(5, 5); p.setLocation(p.x * 2, p.y);`, what are `p`, `q == p`, and `q.distance(30, 29)`?

`Point q = p;` does not create a new point. Both variables refer to the same object. `q.translate(5, 5)` moves this shared object to `(15, 25)`. `p.setLocation(15 * 2, 25)` moves the object to `(30, 25)`. Because `p` and `q` are the same object, `q == p` is `true`. Between `(30, 25)` and `(30, 29)` there is a vertical line: the distance is `4.0`. If an independent copy is wanted, write `new Point(p)`.

```
java.awt.Point[x=30,y=25] true
4.0
140 70 35
```

The third line of the output relates to the next class: `new Color(200, 100, 50).darker()` gives the components of a color.

9.3.2 The `Color` class

`java.awt.Color` holds the color of the interface and of drawings. Every color on the screen is a mixture of three primary colors: red, green, and blue. This scheme is called RGB.

> **Definition 9.9**
>
> An **RGB color** is a triple in which the red, green, and blue components are each an integer in the range 0–255. `new Color(int red, int green, int blue)` creates a color object from this triple. `(0, 0, 0)` is black and `(255, 255, 255)` is white.

There are two ways to obtain a color. The first is the predefined constants, such as `Color.RED`, `Color.GREEN`, `Color.WHITE`, and `Color.LIGHT_GRAY`. These constants also have lowercase aliases (`Color.red`); both refer to the same object. The second is to create a new object from an RGB triple. A value outside 0–255 throws `IllegalArgumentException`; the message of the call `new Color(300, 0, 0)` is `Color parameter outside of expected range: Red`.

```
Color orange = new Color(255, 128, 0);
System.out.println(orange);
System.out.println(orange.getGreen());
System.out.println(Color.RED.darker());
```

```
java.awt.Color[r=255,g=128,b=0]
128
java.awt.Color[r=178,g=0,b=0]
```

Two methods produce darker and lighter shades of a color. `darker()` multiplies each component by 0.7 and truncates to an integer. `brighter()` divides each component by 0.7 and caps the result at 255. Neither method changes the original color; each returns a new `Color` object. `Color` objects are immutable.

Expression	RGB	Calculation
`Color.RED`	(255, 0, 0)	predefined constant
`Color.BLUE`	(0, 0, 255)	predefined constant
`Color.LIGHT_GRAY`	(192, 192, 192)	predefined constant
`new Color(255, 128, 0)`	(255, 128, 0)	orange
`orange.darker()`	(178, 89, 0)	255 × 0,7 = 178,5 → 178; 128 × 0,7 = 89,6 → 89

Expression	RGB	Calculation
`Color.RED.darker()`	(178, 0, 0)	0 × 0,7 = 0
`Color.GREEN.darker()`	(0, 178, 0)	the background of the no-entry sign
`new Color(100, 50, 0).brighter()`	(142, 71, 0)	100 / 0,7 = 142,8 → 142; 50 / 0,7 = 71,4 → 71
`Color.RED.brighter()`	(255, 0, 0)	255 is already at the limit; components that are 0 stay 0
`Color.RED.darker().brighter()`	(254, 0, 0)	178 / 0.7 = 254.2 → 254: truncation loss

Table 9.6: Some colors and their RGB components (verified by running them).

The last row of the table carries a warning. `darker()` and `brighter()` are not exact inverses of each other. Each operation truncates the result to an integer, and the truncated fraction does not come back. If you darken red and then brighten it again, you get `(254, 0, 0)`, not `(255, 0, 0)`.

You may also ask why `brighter()` leaves zero components unchanged. Zero divided by 0.7 is still zero. So pure red does not turn pink when it is "brightened"; it stays the same. The library handles one special case separately: when black `(0, 0, 0)` is brightened it becomes `(3, 3, 3)`, because otherwise black could never be brightened at all.

Example 9.5

Which components does `new Color(200, 100, 50).darker()` give?

Each component is multiplied by 0.7 and truncated: 200 × 0.7 = 140, 100 × 0.7 = 70, 50 × 0.7 = 35. The result is `(140, 70, 35)`. These values are the same as the third line of the previous example's output.

9.3.3 The `Font` class

`java.awt.Font` sets the typeface, style, and size of text. The constructor takes three parameters: `new Font(name, style, size)`.

Parameter	Values
`name`	`"Serif"`, `"SansSerif"`, `"Monospaced"`, `"Dialog"` (logical names available on every system), or a real font such as `"Arial"`
`style`	`Font.PLAIN` (0), `Font.BOLD` (1), `Font.ITALIC` (2); both at once: `Font.BOLD | Font.ITALIC`
`size`	An integer size in points, for example 24

Table 9.7: The parameters of the `Font` constructor.

A logical name is a name that Java maps to a real font on every system. `"Serif"` often corresponds to Times New Roman on Windows, and `"Monospaced"` to Courier New. If a real font name is given and that font is not installed, Java falls back to `"Dialog"`. Portable programs therefore prefer logical names.

The style constants are bit flags. `BOLD` uses bit 1 of the number and `ITALIC` uses bit 2. The `|` operator (bitwise OR) combines two flags: `1 | 2 = 3`. For the object `new Font("Serif", Font.BOLD | Font.ITALIC, 20)`, `getStyle()` returns 3; `isBold()` and `isItalic()` are both `true`.

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(new Color(0, 102, 204));
    g.setFont(new Font("Serif", Font.BOLD, 24));
    g.drawString("Java", 20, 40);
    g.setFont(new Font("Monospaced", Font.PLAIN, 14));
    g.drawString("Swing", 20, 70);
}
```

The method first switches the pen to the blue `(0, 102, 204)`. The first text is written at `(20, 40)` in bold 24-point Serif. Then the font is changed, and the second text is written at `(20, 70)` in 14-point Monospaced. Since the color is not changed, both texts are in the same blue.



Figure 9.11: The output of `FontPanel`: "Java" in bold Serif 24 point and "Swing" in Monospaced 14 point.

The same `Font` object is also used in components such as `JLabel`: `label.setFont(f)`. To derive a new font from an existing one, there is the `deriveFont` method. `f.deriveFont(Font.ITALIC, 18f)` returns a new object in the same family, italic and 18 points. `f` does not change.

Common mistake

Writing `f.deriveFont(18)` to change only the size. The version of `deriveFont` with a single `int` parameter changes the **style**, not the size. For a bold 24-point font, this call interprets 18 as a style and leaves the size at 24. When tried, the result is style 0, size 24. For the size, use the `float` version: `f.deriveFont(18f)`.

9.4 Layout managers

9.4.1 What is a layout manager?

Suppose five buttons are to be put in a window. The position and size of each button could be given by hand: “the first button is at `(10, 10)`, 80 × 30.” This approach runs into a problem. When the user enlarges the window, the buttons stay in the top-left corner. When the screen resolution or the font changes, the button texts no longer fit in the boxes. Java hands this job to an object.

Definition 9.10

A **layout manager** is an object that computes the position and size of the components inside a container according to rules. It is given to the container with `setLayout`: `panel.setLayout(new GridLayout(2, 3));`.

A layout manager is used instead of giving coordinates by hand. When the window grows, or the font or the screen resolution changes, the layout is recomputed automatically. The programmer only chooses the rule: “arrange row by row,” “place in five regions,” “divide into equal cells.” This chapter covers four of Java's standard managers.

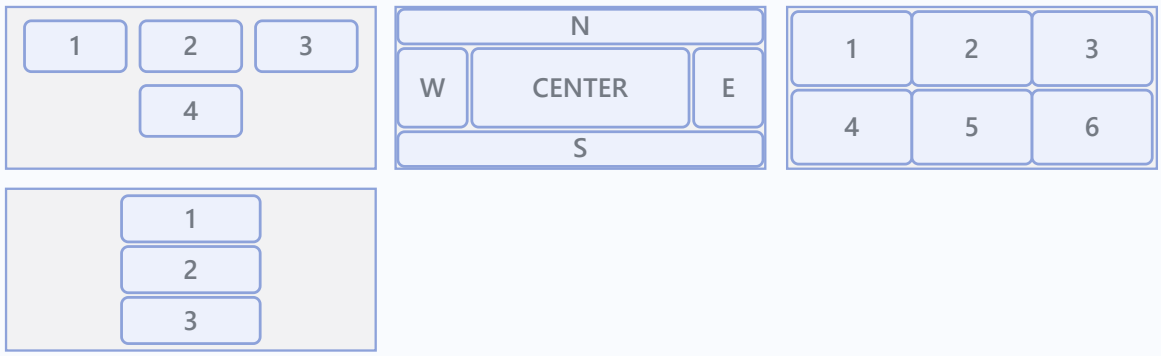


Figure 9.12: Four layout managers. From left to right: `FlowLayout` (left to right, wrapping to the next row when full; the `JPanel` default), `BorderLayout` (five regions; the default of the `JFrame` content pane), `GridLayout(2, 3)` (cells of equal size), `BoxLayout.Y_AXIS` (a single column).

The more flexible but more complicated `GridBagLayout` builds a grid of cells of different sizes. You rarely need it. It is enough to nest panels and give each one a simple manager. For example, the content pane of a window uses `BorderLayout`, while the button strip at the bottom is a separate panel that uses `FlowLayout`.

The behavior of layout managers can be followed interactively in the Swing Layout Managers simulator. As the window size changes, you can see how each of the four managers recomputes its layout.

9.4.2 Code trace: how `FlowLayout` fills rows

The best way to understand how a layout manager works is to compute a layout by hand. The following code adds five buttons of different widths to a panel 260 pixels wide and prints the computed positions.

```
int[] w = {90, 60, 110, 70, 80};
JPanel panel = new JPanel();   // FlowLayout(CENTER, 5, 5)
panel.setSize(260, 120);
for (int i = 0; i < w.length; i++) {
    JButton b = new JButton("B" + i);
    b.setPreferredSize(new Dimension(w[i], 30));
    panel.add(b);
}
panel.doLayout();              // compute positions
for (Component c : panel.getComponents())
    System.out.println(c.getX() + ", " + c.getY());
```

```
52, 5
147, 5
37, 40
152, 40
90, 75
```

The default settings of `FlowLayout` are `FlowLayout(CENTER, 5, 5)`: rows are centered, and a gap of 5 pixels is left between components horizontally and vertically. The same gap is left at the edges of the panel. The preferred size of each button is given with `setPreferredSize`: the widths are 90, 60, 110, 70, and 80; the heights are 30.

Step	Line	Variables	New output	Explanation

1	1: `int[] w = {90, 60, 110, 70, 80};`
	2: `JPanel panel = new JPanel(); // FlowLayout(CENTER, 5, 5)`
	3: `panel.setSize(260, 120);`
		w=[90, 60, 110, 70, 80], panel=260×120	—	The default manager of `JPanel` is `FlowLayout`: centered, with a 5-pixel gap between components. The dashed lines show the margin.

2	5: `JButton b = new JButton("B" + i);` 6: `b.setPreferredSize(new Dimension(w[i], 30));` 7: `panel.add(b);`	i=0, b=B0 (90×30)	—	`add` only appends the button to a list; its position is still (0, 0) and its size 0 × 0.

Step	Line	Variables	New output	Explanation
3	4: `for (int i = 0; i < w.length; i++) {` 5: `JButton b = new JButton("B" + i);` 6: `b.setPreferredSize(new Dimension(w[i], 30));` 7: `panel.add(b);`	i=4, b=B4 (80×30)	—	The loop is over: five buttons were added, and none has been placed. Placement happens only when the layout is computed.
4	9: `panel.doLayout(); // compute positions`	row=1, width=90+5+60=155	—	`doLayout()`: the available width is 260 − 2 × 5 = 250. B0 and B1 fit: 155 pixels.
5	9: `panel.doLayout(); // compute positions`	row=1, trial=155+5+110 = 270 > 250	—	If B2 is added too, the row would be 270 pixels: it does not fit and goes to the **next row**.
6	9: `panel.doLayout(); // compute positions`	row=1, x0=5+(250−155)/2 = 52	—	Row 1 is centered: B0 x = 52, B1 x = 52 + 90 + 5 = 147, y = 5.
7	9: `panel.doLayout(); // compute positions`	row=2, width=110+5+70 = 185	—	Row 2: B2 and B3 (185 pixels); with B4 it would be 270. Centering: x0 = 5 + (250 − 185)/2 = 37.
8	9: `panel.doLayout(); // compute positions`	row=3, x0=5+(250−80)/2 = 90	—	B4 is alone in the third row; y = 5 + 30 + 5 + 30 + 5 = 75.
9	10: `for (Component c : panel.getComponents())` 11: `System.out.println(c.getX() + ", " + c.getY());`	c=B0 … B4	`52, 5 ⏎ 147, 5 ⏎ 37, 40 ⏎ 152, 40 ⏎ 90, 75`	The computed bounds are printed. If the panel gets wider, the rows are recomputed and the buttons move.

Table 9.8: Layout trace of `FlowLayout`.

The first lesson of the trace is in steps 2 and 3. `add` appends the button to the container's component list but does not place it. When the loop ends, all five buttons are at `(0, 0)` with size 0 × 0. Placement happens only when the layout is computed. In a real window, this is triggered automatically by `pack()`, `setVisible(true)`, or a change of window size. In the example, `doLayout()` is called explicitly.

`doLayout()` fills the rows as follows. The available width is the panel width minus the two edge gaps: 260 − 2 × 5 = 250. Buttons are added to a row in order. B0 (90) and B1 (60), with the 5-pixel gap between them, take 155 pixels; they fit. If B2 were added too, the row would be 155 + 5 + 110 = 270 pixels. Since 270 > 250, B2 goes to the next row.

When a row is closed, it is centered. The unused width of the row is distributed equally to both sides: `x0 = 5 + (250 − 155) / 2 = 5 + 47 = 52`. Integer division truncates 47.5 to 47. B0 is placed at `x = 52` and B1 at `x = 52 + 90 + 5 = 147`; both have `y = 5`.

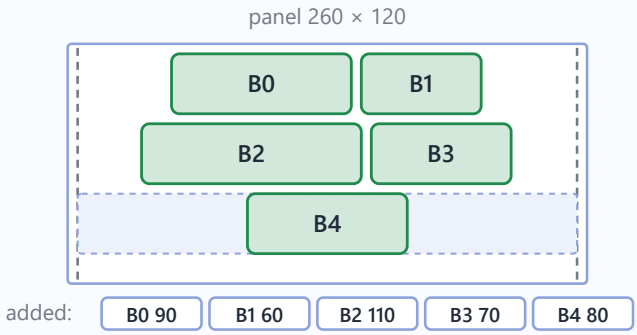


Figure 9.13: Step 8. Three rows have been built; B4 is centered alone in the third row.

The second row, with B2 (110) and B3 (70), takes 185 pixels. If B4 were added, it would be 185 + 5 + 80 = 270; again it does not fit. Centering: `x0 = 5 + (250 − 185) / 2 = 5 + 32 = 37`. B2 is placed at `x = 37` and B3 at `x = 37 + 110 + 5 = 152`. The `y` of the row is the bottom of the previous row plus the gap: `5 + 30 + 5 = 40`. B4 is alone in the third row: `x0 = 5 + (250 − 80) / 2 = 90`, `y = 5 + 30 + 5 + 30 + 5 = 75`. The calculation matches the output exactly.

If the panel gets wider, the rows are recomputed. When the same panel is widened to 400 pixels and the layout is recomputed, the available width becomes 390. The first four buttons (90 + 60 + 110 + 70 + 3 × 5 = 345) fit in one row; B4 goes to the second row:

```
(27, 5) (122, 5) (187, 5) (302, 5) (160, 40)
```

Example 9.6

Three components of size 80 × 30 are added to a `FlowLayout` panel 200 pixels wide. What are the positions of the components?

The available width is 200 − 10 = 190. The first two components take 80 + 5 + 80 = 165 pixels; if the third were added, it would be 250, which does not fit. The start of the first row is `5 + (190 − 165) / 2 = 5 + 12 = 17`. The components are placed at `(17, 5)` and `(102, 5)`. The third component is in the second row: `x = 5 + (190 − 80) / 2 = 60`, `y = 40`. The program run in headless mode gave the same values.

9.4.3 `BorderLayout`

`BorderLayout` divides the container into five regions: `NORTH`, `SOUTH`, `WEST`, `EAST`, and `CENTER`. It is suitable for showing components with different priorities in an orderly way: the toolbar at the top, the status line at the bottom, and the main content in the middle.

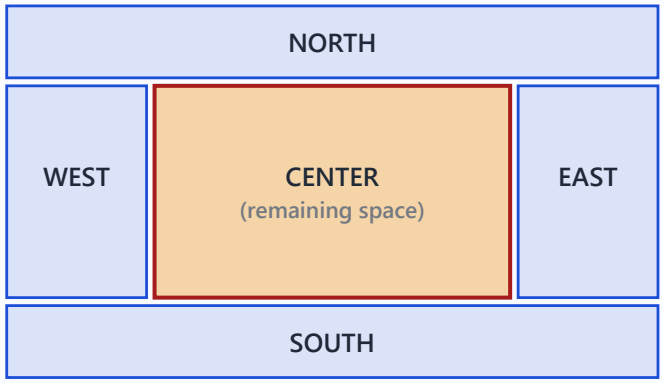


Figure 9.14: The regions of `BorderLayout`.

The manager's rules are as follows:

- A component added without a region goes to the center: `add(b)` and `add(b, BorderLayout.CENTER)` are the same.
- Each region holds a single component. A second addition to the same region replaces the first.
- North and south take the whole width; their height is the component's preferred height. West and east take the height left between north and south; their width is the preferred width. The center fills everything that remains.
- `setHgap()` and `setVgap()` set the horizontal and vertical gap between regions in pixels. The same values can also be given to the constructor: `new BorderLayout(5, 5)`.

The second rule has a subtle side. When a second component is added to the same region, the first is not removed from the container. The container's component count is 2, but the manager attaches only the last one added to that region. The first component takes no part in the layout and keeps the size 0 × 0. It is not visible on the screen.

Common mistake

Adding two components to a window without specifying a region: `f.add(new JButton("One"));` `f.add(new JButton("Two"));`. Because the content pane uses `BorderLayout`, both go to the center and the last one added wins. Only “Two” is visible, and it fills the whole area. Tried on a 300 × 200 `BorderLayout` panel, “Two” comes out as `(0, 0, 300, 200)` and “One” as 0 × 0. Side-by-side arrangement is the behavior of `FlowLayout`; for that, put the buttons in a `JPanel` first.

9.4.4 Code trace: filling the regions

```
JPanel panel = new JPanel();
BorderLayout layout = new BorderLayout();
layout.setHgap(5);
layout.setVgap(5);
panel.setLayout(layout);
panel.add(new JButton("North"), BorderLayout.NORTH);
panel.add(new JButton("South"), BorderLayout.SOUTH);
panel.add(new JButton("West"), BorderLayout.WEST);
panel.add(new JButton("East"), BorderLayout.EAST);
panel.add(new JButton("Center"), BorderLayout.CENTER);
```

The code creates a panel and a `BorderLayout` object, sets the gaps to 5 pixels, and gives the manager to the panel. Then a button is added to each of the five regions. Table 9.9 describes the effect of each addition on the layout.

Step	Line	Explanation
1	1: `JPanel panel = new JPanel();` 2: `BorderLayout layout = new BorderLayout();` 3: `layout.setHgap(5);` 4: `layout.setVgap(5);` 5: `panel.setLayout(layout);`	The panel will use `BorderLayout`; there is a 5-pixel gap between regions. There are no components yet.
2	6: `panel.add(new JButton("North"), BorderLayout.NORTH);`	The North button takes the **whole width**; its height is the button's preferred height.
3	7: `panel.add(new JButton("South"), BorderLayout.SOUTH);`	South also takes the whole width and sits at the very bottom.
4	8: `panel.add(new JButton("West"), BorderLayout.WEST);`	West takes the height left between north and south; its width follows its text.
5	9: `panel.add(new JButton("East"), BorderLayout.EAST);`	East sits at the right edge.
6	10: `panel.add(new JButton("Center"), BorderLayout.CENTER);`	Center fills **all the remaining space**. If the window is enlarged, the center grows the most.

Table 9.9: Filling the regions of `BorderLayout`.

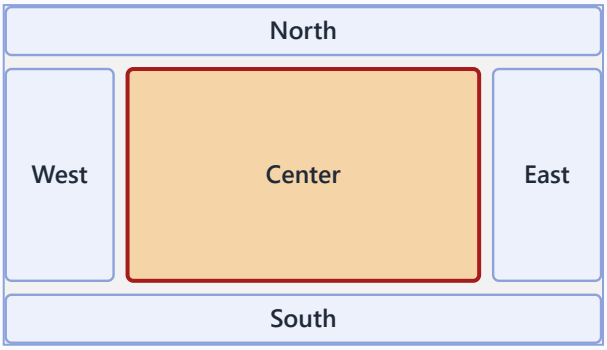


Figure 9.15: Step 6. All five regions are filled; the center has taken all the remaining space.

The pixel values depend on the panel size and on the buttons' preferred sizes. If the panel is set to 400 × 250 and the layout is computed, the values come out as follows. In the default font the buttons are 26 pixels high. North is 400 × 26 at `(0, 0)`, and south is 400 × 26 at `(0, 224)`. West and east start below north with a 5-pixel gap: `y = 26 + 5 = 31`. Their height is `250 − 26 − 26 − 2 × 5 = 188`. West is 56 pixels wide and east 63; these are the widths that the texts “West” and “East” require. The center starts at `x = 56 + 5 = 61` and is `400 − 56 − 63 − 2 × 5 = 271` pixels wide. The 76 pixels that the center prefers are not used at all in this calculation.

If the window is enlarged, the center grows the most. North and south only widen, and west and east only lengthen. This property makes `BorderLayout` suitable for the main window layout: a text area or a drawing canvas is put in the center and grows with the window.

Example 9.7

Five components are added to a 300 × 200 pixel `BorderLayout` panel with no gaps. The preferred sizes are: north 100 × 30, south 100 × 20, west 60 × 50, east 40 × 50, center 10 × 10. Then a second component with preferred size 80 × 40 is added to the north. What are the bounds of the components?

North takes the whole width; its height is the preferred 30: `(0, 0, 300, 30)`. The preferred width of 100 is ignored. South is at the bottom: `y = 200 − 20 = 180`, so `(0, 180, 300, 20)`. West and east take the height between them, `200 − 30 − 20 = 150` pixels: west `(0, 30, 60, 150)`, east `(300 − 40, 30, 40, 150) = (260, 30, 40, 150)`. The center takes the rest: `(60, 30, 200, 150)`. The second north component replaces the first and is 40 pixels high. West, east, and center move down 10 pixels and become shorter: west `(0, 40, 60, 140)`, center `(60, 40, 200, 140)`. The old north component stays in the panel, but its size is not updated. The program run in headless mode gave all these values.

9.4.5 `GridLayout` and `BoxLayout`

`GridLayout(rows, columns)` divides the container into cells of equal size. Components are placed row by row, from left to right. Each component completely fills its cell; the preferred size is ignored.

Example 9.8

Five buttons are added in order to a 300 × 120 pixel panel that uses `new GridLayout(2, 3)`. What are the position and size of the fifth button?

The cell width is 300 / 3 = 100 and the height is 120 / 2 = 60. Buttons 1–3 fill the top row. Button 4 goes into the first cell of the bottom row and button 5 into the second cell: `(100, 60, 100, 60)`. The sixth cell stays empty. The cell size does not change with the number of components; the program run gave the same bounds.

One of the row or column counts may be given as 0. `new GridLayout(0, 2)` means "two columns, as many rows as needed." If five buttons are added to a 200 × 150 panel, three rows are formed. The cells are 100 × 50, and the fifth button is placed at `(0, 100)`. If the number of components exceeds rows × columns and neither is zero, Java keeps the number of rows and increases the number of columns. When seven buttons are added to `GridLayout(2, 3)`, the grid becomes 2 × 4 and the cells narrow to 75 pixels.

`BoxLayout` arranges components in a single row or a single column. Its constructor also takes the container it manages as a parameter: `panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS))`. `Y_AXIS` stacks components vertically and `X_AXIS` places them side by side. It is used in forms to line up labels and fields one under another.

Common mistake

Setting a component's position with `setBounds` or `setLocation` in a container that has a layout manager. The manager overwrites these values at the next layout computation, and the position you set is lost. To influence a component's size, use `setPreferredSize`. The manager takes this preference into account or not, depending on its rule.

9.5 Application: the no-entry sign

9.5.1 Goal and layers

This application writes a panel that draws the "no entry" traffic sign. The sign consists of four layers painted on top of one another:

1. A green background (with `setBackground`).

2. A large white circle (the outer ring).
3. A red circle, slightly smaller.
4. A white horizontal bar in the middle.

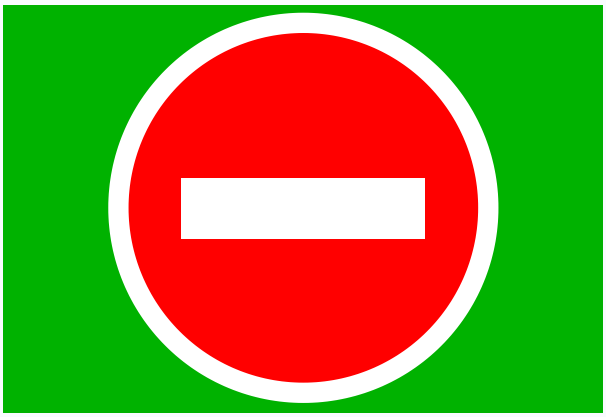


Figure 9.16: The no-entry sign: a white ring, a red circle, and a white horizontal bar in the middle on a green background.

The program makes one important design decision: all measurements are computed from `getWidth()` and `getHeight()`. No fixed pixel values are used. When the window is enlarged, the sign grows with it and always stays in the center. The drawing is recomputed from scratch each time, according to the current size of the panel.

9.5.2 The `paintComponent` code

```
public class NoEntrySign extends JPanel {
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        var c = new Point(getWidth() / 2, getHeight() / 2); // center
        var r = Math.min(getWidth() / 2, getHeight() / 2) - 5; // radius
        var r2 = (int) (r * 0.9);   // inner radius
        var bw = (int) (r2 * 1.4);  // bar width
        var bh = (int) (r2 * 0.35); // bar height
        g.setColor(Color.WHITE);
        g.fillOval(c.x - r, c.y - r, 2 * r, 2 * r);
        g.setColor(Color.RED);
        g.fillOval(c.x - r2, c.y - r2, r2 * 2, r2 * 2);
        g.setColor(Color.WHITE);
        g.fillRect(c.x - bw / 2, c.y - bh / 2, bw, bh);
    }
}
```

The code computes five variables and then draws three shapes. The variables are declared with `var`. Since Java 10, `var` lets the compiler infer the type of a local variable from the expression on the right. Here the type of `c` is `Point` and the others are `int`. The classic form is `Point c = new Point(…);` `int r = …;`. `var` does not remove the type; the type is determined at compile time and never changes afterward.

Let us read the lines in order:

- `c`: The center of the panel. The width and height are divided by two. Since both operands are `int`, integer division is used.
- `r`: The radius of the outer circle. The distance from the center to the shorter side is chosen (`Math.min`), and a margin of 5 pixels is left. This way the circle does not spill outside the panel.
- `r2`: The radius of the inner circle, 90% of `r`. `r * 0.9` is a `double`; the `(int)` cast discards the fractional part.
- `bw` and `bh`: The width and height of the bar, 1.4 and 0.35 times the inner radius.

The drawing is done in three steps. First the pen is switched to white and the outer circle is drawn. Then the pen is switched to red and the inner circle is drawn. The red circle is painted on top of the white one, so only a thin ring of white remains on the outside. Finally the pen is switched back to white and the bar is drawn. The bar is given by the top-left corner of its bounding rectangle: from the center, half the width to the left and half the height up.

The necessary imports are `java.awt.*` and `javax.swing.*`. `Point`, `Graphics`, and `Color` come from the first, and `JPanel` from the second.

9.5.3 Code trace: drawing the sign

Let the panel be 384 × 261 pixels. This size is typical of the inner area left from a 400 × 300 window after the title bar and borders are subtracted. The real value varies by a few pixels depending on the operating system and theme settings.

Step	Line	Variables	Explanation
1	1: `super.paintComponent(g);`	getWidth()=384, getHeight()=261	The panel is 384 × 261 pixels. `super.paintComponent` paints the background with the background color (green).
2	2: `var c = new Point(getWidth() / 2, getHeight() / 2);`	getWidth()=384, getHeight()=261, c=(192, 130)	Center: (384 / 2, 261 / 2) = (192, 130). Integer division: 130.5 → 130.
3	3: `var r = Math.min(getWidth() / 2, getHeight() / 2) - 5;`	getWidth()=384, getHeight()=261, c=(192, 130), r=125	Radius: min(192, 130) − 5 = 125. It is chosen by the shorter side, leaving a 5-pixel margin.
4	4: `var r2 = (int) (r * 0.9);`	getWidth()=384, getHeight()=261, c=(192, 130), r=125, r2=112	Inner radius: (int) 112.5 = 112.

Step	Line	Variables	Explanation
5	5: `var bw = (int) (r2 * 1.4);`	getWidth()=384, getHeight()=261, c=(192, 130), r=125, r2=112, bw=156	Bar width: (int) (112 × 1.4) = (int) 156.8 = 156.
6	6: `var bh = (int) (r2 * 0.35);`	getWidth()=384, getHeight()=261, c=(192, 130), r=125, r2=112, bw=156, bh=39	Bar height: (int) 39.2 = 39.
7	7: `g.setColor(Color.WHITE);` 8: `g.fillOval(c.x - r, c.y - r, 2 * r, 2 * r);`	getWidth()=384, getHeight()=261, c=(192, 130), r=125, r2=112, bw=156, bh=39	`fillOval(67, 5, 250, 250)`: the white circle; its bounding square is the dashed line.
8	9: `g.setColor(Color.RED);` 10: `g.fillOval(c.x - r2, c.y - r2, r2 * 2, r2 * 2);`	getWidth()=384, getHeight()=261, c=(192, 130), r=125, r2=112, bw=156, bh=39	`fillOval(80, 18, 224, 224)`: the red circle is painted over the white one; a white ring 13 pixels thick remains.
9	11: `g.setColor(Color.WHITE);` 12: `g.fillRect(c.x - bw / 2, c.y - bh / 2, bw, bh);`	getWidth()=384, getHeight()=261, c=(192, 130), r=125, r2=112, bw=156, bh=39	`fillRect(114, 111, 156, 39)`: the white bar on top. The sign is complete.

Table 9.10: Trace of `paintComponent` for a 384 × 261 panel.

In step 2, the center `(384 / 2, 261 / 2)` is computed. 261 / 2 is 130.5 in mathematics, but integer division discards the fraction: `(192, 130)`. In step 3, the radius is found as `min(192, 130) − 5 = 125`. The shorter side is the height, so it determines the radius.

Steps 4–6 show type conversion. `125 × 0.9 = 112.5`; `(int)` truncates it to 112. There is no rounding: even 112.5 goes down. `112 × 1.4` should be 156.8. `double` arithmetic cannot represent this value exactly; in Java the result of the product is `156.79999999999998`. Truncation still gives 156. The bar height is `112 × 0.35 = 39.2`; truncated, it is 39.

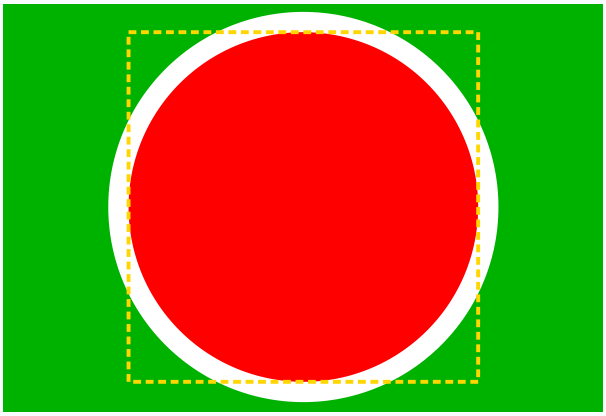


Figure 9.17: Step 8. `fillOval(80, 18, 224, 224)` : the red circle has been painted over the white one; the bounding square is shown with a dashed line.

The arguments of the call `fillOval(c.x - r, c.y - r, 2 * r, 2 * r)` in step 7 are `(192 - 125, 130 - 125, 250, 250) = (67, 5, 250, 250)`. In step 8 the inner circle is drawn with `(192 - 112, 130 - 112, 224, 224) = (80, 18, 224, 224)`. The difference between the two radii, 125 − 112 = 13, gives the 13-pixel white ring. In step 9 the bar is drawn at `(192 - 78, 130 - 19, 156, 39) = (114, 111, 156, 39)`. The values `bw / 2 = 78` and `bh / 2 = 19` are also integer divisions.

The computed calls were verified in Java. The panel was set to 384 × 261 in headless mode and `paintComponent` was run on an image with a `DebugGraphics` pen. The log showed exactly `fillOval(67, 5, 250, 250)`, `fillOval(80, 18, 224, 224)`, and `fillRect(114, 111, 156, 39)`. The pixels in the image also had the expected colors: the center is white (the bar), `(192, 25)` is red, `(192, 10)` is white (the ring), and `(5, 5)` is green.

Example 9.9

If the same `paintComponent` runs on a 240 × 180 pixel panel, what are the variables and the arguments of the three drawing calls?

The center is `(120, 90)`. `r = min(120, 90) - 5 = 85`. `r2 = (int) (85 × 0.9) = (int) 76.5 = 76`. `bw = (int) (76 × 1.4) = (int) 106.4 = 106`; `bh = (int) (76 × 0.35) = (int) 26.6 = 26`. Truncation lowers even 26.6 to 26. The calls are as follows:

```
setColor(0, 178, 0)
fillRect(0, 0, 240, 180)
setColor(255, 255, 255)
fillOval(35, 5, 170, 170)
setColor(255, 0, 0)
fillOval(44, 14, 152, 152)
setColor(255, 255, 255)
fillRect(67, 77, 106, 26)
```

The outer circle is `(120 - 85, 90 - 85, 170, 170)` and the inner circle is `(120 - 76, 90 - 76, 152, 152)`; the ring is 9 pixels thick. For the bar, `bw / 2 = 53` and `bh / 2 = 13`: `(67, 77, 106, 26)`.

Example 9.10

If the panel were 200 × 100 pixels, what would `r`, `r2`, `bw`, and `bh` be?

`r = min(100, 50) - 5 = 45`. `r2 = (int) 40.5 = 40`. `bw = (int) (40 × 1.4)` and `bh = (int) (40 × 0.35)`. A floating-point trap may come to mind here: `1.4` and `0.35` cannot be represented exactly in binary, so could the product come out as 55.999… and be truncated to 55? Tried in Java, `40 * 1.4` gave exactly `56.0` and `40 * 0.35` gave exactly `14.0`. The result is `bw = 56`, `bh = 14`. In boundary cases like this, the way to trust the result is to run the calculation.

Common mistake

Mixing up the order of the layers. If the red circle is drawn before the white one, the large white circle covers it completely and the sign turns into a white disk. If the bar is drawn before the red circle, the red circle covers the bar. Whatever is drawn later is always on top; the drawing order is the layer order.

9.5.4 `main` and the window

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
        var panel = new NoEntrySign();
        panel.setBackground(Color.GREEN.darker());
        var frame = new JFrame("A simple graphics program");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(panel, BorderLayout.CENTER);
        frame.setVisible(true);
    });
}
```

The only job of the `main` method is to hand the code that builds the interface to `SwingUtilities.invokeLater`. `() -> { … }` is a lambda expression: “a piece of code to be run later.” `invokeLater` puts this piece in a queue to be run on the event dispatch thread. This pattern appears in the `main` of every Swing program; the reason lies in how the event dispatch thread works.

Inside the lambda, the panel is created and its background is set to `Color.GREEN.darker()`, that is, `(0, 178, 0)`. Pure green `(0, 255, 0)` is bright enough to strain the eyes; the darker shade looks more like a traffic sign. `super.paintComponent(g)` paints the background with this color. The panel is added to the center of the window's content pane. Since it is the only component, it fills the whole area.

In a 400 × 300 window, after the border and title bar are subtracted, the panel is about 384 × 261. This value varies with the operating system. The program's correctness does not depend on it: the measurements are recomputed from `getWidth()` and `getHeight()` at every painting. If the panel is enlarged to 768 × 522, the outer circle is drawn with `fillOval(128, 5, 512, 512)` and the sign is still in the center.

9.6 Mouse events and the event dispatch thread

9.6.1 Interactive graphics and the event loop

Computer graphics is the process of creating and manipulating visual content. In interactive graphics, the user changes the image with input devices such as the mouse and keyboard. Drawing programs, games, and maps are of this kind. Graphics libraries provide drawing and coloring shapes; interaction is built with an event loop.

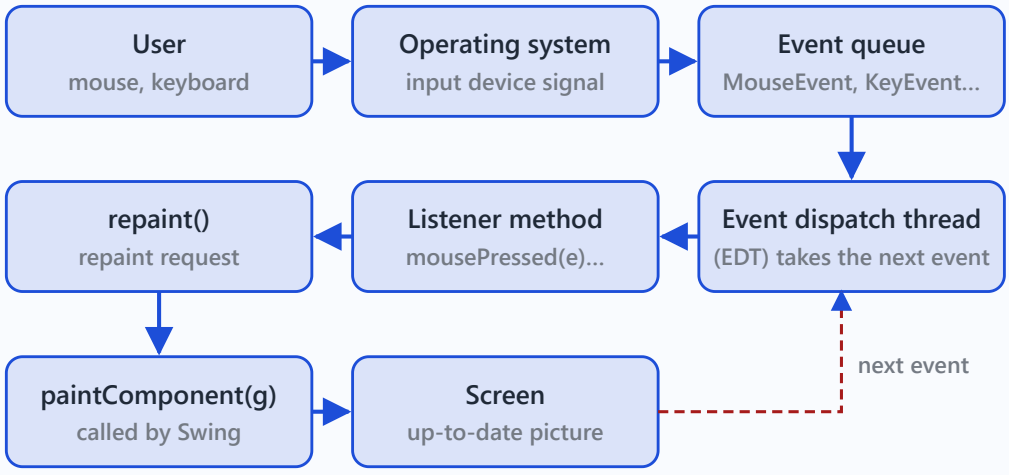


Figure 9.18: The event loop. The user's action passes through the operating system and enters the event queue. The event dispatch thread hands the event to the listener method; the method updates the data and requests a `repaint()`. At a suitable moment Swing calls `paintComponent` and the screen is updated.

The steps in the loop are as follows. The user presses the mouse. The operating system detects this action and notifies Java. Java creates an event object and puts it in the event queue. The event dispatch thread takes the next event in the queue and calls the method of the listener registered with that

component. The listener method updates the program's data and requests a redraw with `repaint()`. At a suitable moment Swing calls `paintComponent`, and the screen is drawn with the new data. The event dispatch thread moves on to the next event.

This loop has a rule: a listener method does not paint the screen directly. It only changes the data and requests a repaint. The drawing is always done by `paintComponent`. This separation keeps the drawing in one place and ensures that the correct picture appears every time the window is repainted.

9.6.2 Listener interfaces and adapter classes

Definition 9.11

An **event listener** is an interface that defines the methods to be called when an event occurs, or an object that implements this interface. A listener object is registered with a component using an `add…Listener` method.

For mouse events, Java's interface is `MouseListener`. The interface has five abstract methods. If a class implements this interface, it must write all five methods. The bodies of the methods it is not interested in stay empty.

```
class ClickListener implements MouseListener {
    public void mouseClicked(MouseEvent e) {
        System.out.println("click");
    }
    public void mousePressed(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
}
```

Four of the five methods are empty. For a class interested only in clicks, this is extra typing. Java offers ready-made classes for this situation.

Definition 9.12

An **adapter class** is a ready-made class that implements all the methods of a listener interface with empty bodies. A class derived from it overrides only the method it is interested in.

```
class ClickAdapter extends MouseAdapter {
    @Override
    public void mouseClicked(MouseEvent e) {
        System.out.println("click");
    }
}
// panel.addMouseListener(new ClickAdapter());
```

`MouseAdapter` is an abstract class, but it has no abstract methods; all its methods have empty bodies. The purpose of declaring it abstract is to prevent creating an object of it directly. `MouseAdapter` implements three interfaces at once: `MouseListener`, `MouseMotionListener`, and `MouseWheelListener`. A single subclass can thus handle click, motion, and wheel events.

Motion events are in a separate interface: `MouseMotionListener`. This interface has two methods: `mouseDragged` and `mouseMoved`. Its own adapter is `MouseMotionAdapter`. Registration is separate too: click-type events are registered with `addMouseListener(…)` and motion events with `addMouseMotionListener(…)`.

The listener class does not even need to be written in a separate file. Java allows a single-use class to be defined where it is used.

Definition 9.13

An **anonymous inner class** is an inner class that has no name and whose single object is created where it is defined. The notation `new MouseAdapter() { … }` defines a nameless class derived from `MouseAdapter` and immediately creates an object of it.

Common mistake

Implementing an interface and not writing some of its methods. If `class ClickListener implements MouseListener` writes only `mouseClicked`, the compiler gives this error:

```
ClickListener.java:6: error: ClickListener is not abstract and does not override
abstract method mouseExited(MouseEvent) in MouseListener
class ClickListener implements MouseListener {
```

The compiler names only one of the missing methods. Even if `mousePressed` is written too, the message still points to `mouseExited`; the message is not a list of the missing methods. The solution is either to write all five methods or to derive from `MouseAdapter`.

9.6.3 Mouse event methods

Method	Interface	When is it called?
`mousePressed(e)`	`MouseListener`	The moment a mouse button is pressed
`mouseReleased(e)`	`MouseListener`	When the button is released
`mouseClicked(e)`	`MouseListener`	When pressed and released at the same place (pressed → released → clicked)
`mouseEntered(e)` / `mouseExited(e)`	`MouseListener`	When the pointer enters / leaves the component
`mouseMoved(e)`	`MouseMotionListener`	When it moves without a button pressed
`mouseDragged(e)`	`MouseMotionListener`	When it moves while a button is held down (dragging)
`mouseWheelMoved(e)`	`MouseWheelListener`	When the wheel is turned

Table 9.11: Mouse event methods.

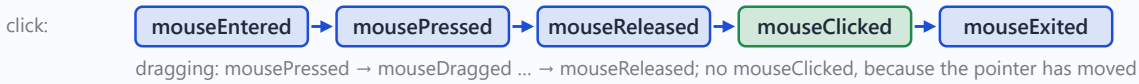


Figure 9.19: The order of mouse events. A click produces three events; a drag produces many `mouseDragged` events between the press and the release.

A click is not a single event. The user presses the button: `mousePressed`. The user releases it: `mouseReleased`. If the press and the release were at the same point, Java also generates a `mouseClicked`. If the mouse moved between the press and the release, no `mouseClicked` arrives; `mouseDragged` events come in between. That is why applications such as a drawing program listen not for clicks but for presses and drags.

When press, release, and click events were sent in turn to a panel in headless mode, the listeners were called in this order. Because both `ClickListener` (which implements the interface) and `ClickAdapter` (derived from the adapter) were registered with the same panel, “click” was printed twice on a click. Several listeners can be registered with one component; when an event arrives, all of them are called.

9.6.4 The `MouseEvent` class

Every mouse method takes a `MouseEvent e` parameter. Where the event happened, with which button, and with which keys held down is all in this object.

Method	Returns
`getX()`, `getY()`	The pointer's position relative to the component

Method	Returns
`getPoint()`	The same position, as a `Point`
`getButton()`	`BUTTON1` left, `BUTTON2` middle, `BUTTON3` right (the values 1, 2, 3)
`getClickCount()`	The number of consecutive clicks (2: double click)
`isShiftDown()`, `isCtrlDown()`, `isAltDown()`	Is the key held down at that moment?

Table 9.12: Frequently used methods of the `MouseEvent` class.

```
panel.addMouseListener(new MouseAdapter() {
    @Override
    public void mousePressed(MouseEvent e) {
        String button = "left";
        if (e.getButton() == MouseEvent.BUTTON3) {
            button = "right";
        }
        System.out.println(button + " (" + e.getX()
                + ", " + e.getY() + ")"
                + (e.isShiftDown() ? " + Shift" : ""));
    }
});
```

The code registers an anonymous adapter with a panel. If the pressed button is the right button, the variable `button` becomes “right.” Then the button name, the position, and, if the Shift key is held down, “ + Shift” are printed. Running it showed that a left click at `(40, 120)` on the panel prints `left (40, 120)`, and a right click at `(70, 30)` with Shift held down prints `right (70, 30) + Shift`.

Coordinates are relative not to the screen but to the component that received the event. The top-left corner of a panel is `(0, 0)` wherever the panel is in the window. This makes it possible to compare mouse coordinates directly with the shapes drawn inside the panel.

In short, a click produces three events in the order `pressed → released → clicked`; while dragging, many `mouseDragged` events arrive. The right button is `BUTTON3`.

9.6.5 The event dispatch thread (EDT)

The `main` of a Swing program is written with the `SwingUtilities.invokeLater` pattern. The reason for this pattern is Swing's relationship with threads.

Definition 9.14

The **event dispatch thread** (EDT) is the single thread on which Swing components are built and modified and on which events are processed. It runs the tasks in the event queue one at a time, in order. In Java 21 its name is `AWT-EventQueue-0`.

Swing components are not thread-safe. If several threads touch the same component at the same time, the internal state of the component can become inconsistent. Swing solves this problem not with locks but with a simple rule: components are built and modified only from the event dispatch thread. Listener methods already run on the EDT, so it is safe to touch components from inside them.

An analogy explains the rule. In the kitchen there is a single cook: the EDT. The waiters do not cook the food themselves; they leave an order slip. The slip is a `Runnable` object, that is, a piece of code to be run. The cook prepares the slips one at a time, in the order they arrive. Since there is only one cook, two dishes never enter the same pan at the same time.

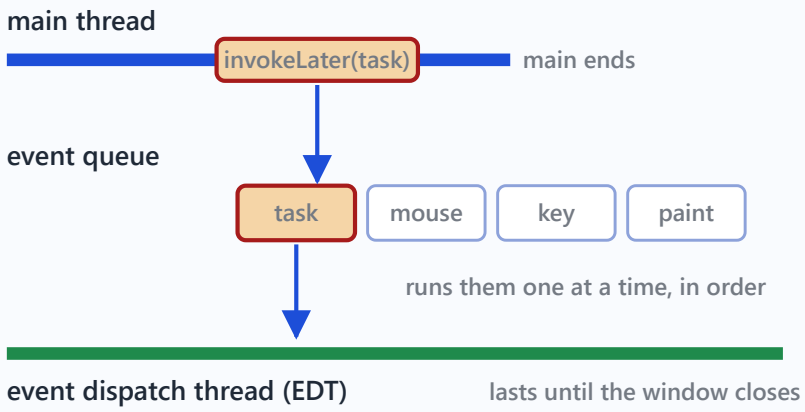


Figure 9.20: The `main` thread puts a task in the event queue with `invokeLater` and finishes. The event dispatch thread runs the tasks in the queue in order.

Method	Behavior
`invokeLater(task)`	Puts the task in the queue and returns immediately (asynchronous). The standard way to start a GUI.
`invokeAndWait(task)`	Waits until the task finishes on the EDT (synchronous). If called from inside the EDT, it throws an `Error`: the EDT cannot wait for itself.

Table 9.13: The two invocation methods of `SwingUtilities`.

Calling `invokeAndWait` from inside the EDT really does throw an `Error`. If the EDT waited for a task it had itself put in the queue, nobody would be left to run that task, and the program would hang forever. Swing catches this in advance:

```
AWT-EventQueue-0 daemon=false
java.lang.Error: Cannot call invokeAndWait from the event dispatcher thread
```

The first line shows which thread the task run with `invokeAndWait` is on: `AWT-EventQueue-0`. This thread is not a daemon (background) thread. The JVM does not exit while a non-daemon thread is running. This is why a GUI program continues after `main` ends.

9.6.6 Code trace: `invokeLater` and `invokeAndWait`

```
public static void main(String[] a) throws Exception {
  System.out.println("1 main");
  SwingUtilities.invokeLater(
      () -> System.out.println("3 task A"));
  SwingUtilities.invokeLater(
      () -> System.out.println("4 task B"));
  System.out.println("2 main goes on");
  SwingUtilities.invokeAndWait(() -> System.out.println(
      "5 task C, EDT? " + SwingUtilities.isEventDispatchThread()));
  System.out.println("6 main, EDT? "
      + SwingUtilities.isEventDispatchThread());
}
```

```
1 main
2 main goes on
3 task A
4 task B
5 task C, EDT? true
6 main, EDT? false
```

Step	Line	Variables	New output
1	2: `System.out.println("1 main");`	thread=main	`1 main`
2	3: `SwingUtilities.invokeLater(` 4: `() -> System.out.println("3 task A"));`	queue=[A]	—
3	5: `SwingUtilities.invokeLater(` 6: `() -> System.out.println("4 task B"));`	queue=[A, B]	—
4	7: `System.out.println("2 main goes on");`	queue=[A, B]	`2 main goes on`
5	8: `SwingUtilities.invokeAndWait(() -> System.out.println(` 9: `"5 task C, EDT? " + SwingUtilities.isEventDispatchThread()));`	queue=[A, B, C]	—
6	4: `() -> System.out.println("3 task A"));`	queue=[B, C]	`3 task A`
7	6: `() -> System.out.println("4 task B"));`	queue=[C]	`4 task B`
8	8: `SwingUtilities.invokeAndWait(() -> System.out.println(` 9: `"5 task C, EDT? " + SwingUtilities.isEventDispatchThread()));`	queue=[]	`5 task C, EDT? true`
9	10: `System.out.println("6 main, EDT? "` 11: `+ SwingUtilities.isEventDispatchThread());`	thread=main	`6 main, EDT? false`

Table 9.14: Trace of the event queue.

The program starts on the `main` thread and prints `1 main`. In step 2, `invokeLater` puts task A in the queue and returns immediately. If the event dispatch thread does not exist yet, this first call starts it. In step 3, task B is also added to the queue; `main` still does not wait. In step 4, `main` continues with its own work and prints `2 main goes on`.



Figure 9.21: Step 5. `invokeAndWait` has put task C in the queue and stopped `main`. A, B, and C are waiting in the queue.

In step 5, `invokeAndWait` puts task C in the queue and stops `main`. The EDT processes the queue from the beginning: first A (`3 task A`), then B (`4 task B`), then C. The queue is ordered; tasks run in the order they entered it. Task C asks which thread it is on with `isEventDispatchThread()` and gets `true`. When C finishes, the waiting `main` is woken up. In the last line, `main` asks the same question and gets `false`.

We must be careful about the order of the output. The order 1, 2, 3, 4, 5, 6 was seen in this run. When the program was run 20 times, the order was the same each time. But part of the order is not guaranteed. The line `2 main goes on` and the line `3 task A` come from different threads. The EDT could have run A before `main` printed line 2. What is certain is this: A runs before B, B before C, and C before 6. The same distinction applies to the sequence `invokeLater(B)`, `println(A)`, `invokeAndWait(C)`, `println(D)`. B comes before C and C before D. In 20 trials `ABCD` was always seen, but the places of A and B may change from run to run.

Example 9.11

A button listener runs `System.out.println("1")`, `SwingUtilities.invokeLater(() -> System.out.println("3"))`, and `System.out.println("2")` in that order. What is the output? Is the order certain?

`1`, `2`, `3`. The listener runs on the EDT. `invokeLater` puts the task at the end of the queue and returns; the listener prints `2` and finishes. Only after that can the EDT take the next task in the queue. This time the order is certain, because both pieces run on the same thread. A task put in the queue cannot start before the task that is currently running.

Common mistake

Building the interface directly on the `main` thread. In small programs this often causes no problem, so the mistake goes unnoticed for a long time. But once the window becomes visible, the EDT starts running and can touch the same components as `main`. The result is rare, random drawing glitches or exceptions. There is only one safe pattern: `SwingUtilities.invokeLater(() -> { … })`.

9.7 Application: a drawing program

This application is a program in which you draw by pressing and dragging the mouse. The principle is summarized in three sentences:

- On a press (`mousePressed`), a new curve starts.
- As the mouse is dragged (`mouseDragged`), points are added to the end of the curve and a repaint is requested.
- When painting (`paintComponent`), the consecutive points of each curve are joined with lines.

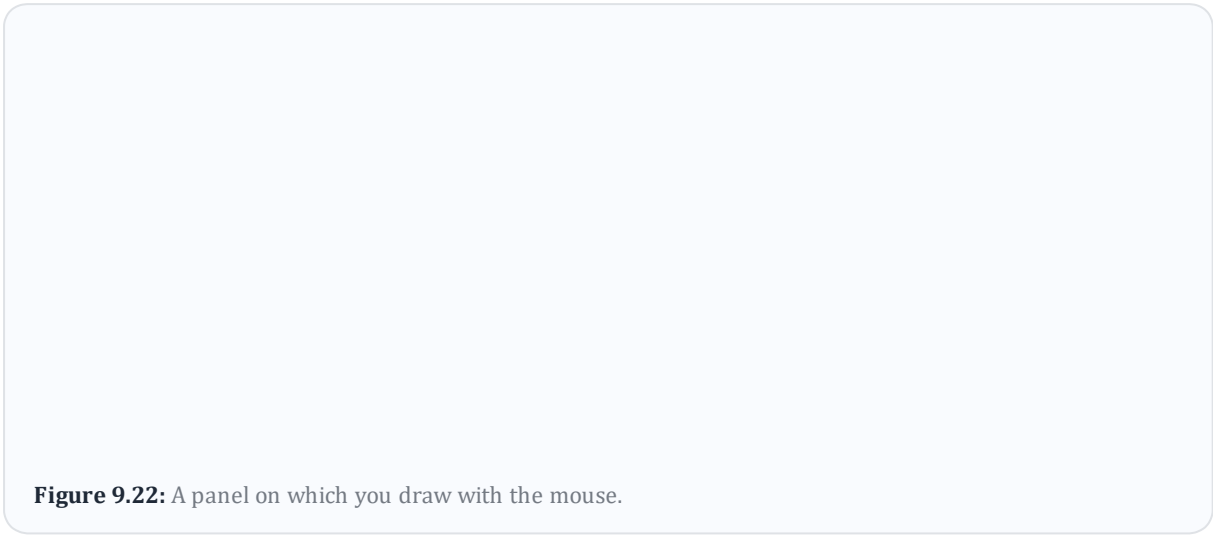


Figure 9.22: A panel on which you draw with the mouse.

The basic design decision of the program is to store the drawing not as an image but as a list of points. The screen is redrawn from this data each time. The drawing is not lost even if the window is minimized and restored, or another window passes over it and is dragged away. Separating the data from the picture applies the rule of the event loop: listeners change the data, and `paintComponent` draws the data.

9.7.1 Storing the curves

```
public class SimpleDrawPanel extends JPanel {
    private List<List<Point>> curves = new ArrayList<>();

    public SimpleDrawPanel() {
        addMouseListener(new MouseAdapter() {
            @Override
            public void mousePressed(MouseEvent e) {
                var curve = new ArrayList<Point>();
                curve.add(new Point(e.getX(), e.getY()));
                curves.add(curve);
            }
        });
        addMouseMotionListener(new MouseMotionAdapter() {
            @Override
            public void mouseDragged(MouseEvent e) {
                curves.get(curves.size() - 1)
                      .add(new Point(e.getX(), e.getY()));
                repaint(0, 0, getWidth(), getHeight());
            }
        });
    }
}
```

The class's only field is `curves`. Its type is `List<List<Point>>`, a list of lists. The outer list holds the curves. Each inner list holds the points of one curve in order.

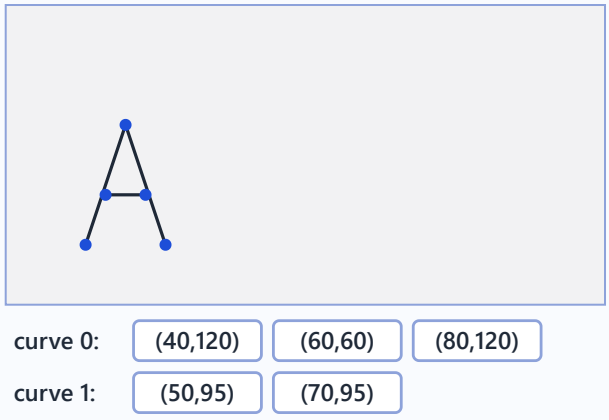


Figure 9.23: A drawing with two curves and the data that holds it. Curve 0 consists of three points and curve 1 of two.

The constructor registers two listeners. The first is an anonymous class derived from `MouseAdapter` that writes only `mousePressed`. The method creates a new, empty list of points, adds the pressed point to the list, and puts the list in the outer list. The second listener is derived from

`MouseMotionAdapter` and writes only `mouseDragged`. The method takes the last element of the outer list, that is, the most recently started curve, and appends the new point to it. Then it has the panel repainted.

The last curve is reached with `curves.get(curves.size() - 1)`. Since list indices start at 0, the index of the last element is `size() - 1`. `get(size())` always asks for one position past the end of the list.

Common mistake

Asking for the last element with `get(size())`. When the list holds one curve, this call throws the following exception:

```
java.lang.IndexOutOfBoundsException: Index 1 out of bounds for length 1
```

The error is not caught at compile time; it appears on the first drag. The correct index is `size() - 1`.

The code has an import trap as well. If `java.awt.*` and `java.util.*` are imported together, the name `List` exists in both packages: `java.awt.List` is a list-box component, whereas `java.util.List` is the collection interface. The compiler cannot tell which one is meant:

```
Main.java:4: error: reference to List is ambiguous
  List<String> x;
  both interface java.util.List in java.util and class java.awt.List in java.awt match
```

The solution is to import the intended class explicitly as well: `import java.util.List;`. A single-type import takes precedence over an on-demand (star) import.

9.7.2 Code trace: mouse events

Suppose the user makes the following moves: presses at `(40, 120)`, drags to `(60, 60)` and `(80, 120)`, and releases; then presses again at `(50, 95)` and drags to `(70, 95)`.

```
addMouseListener(new MouseAdapter() {
    @Override
    public void mousePressed(MouseEvent e) {
        var curve = new ArrayList<Point>();
        curve.add(new Point(e.getX(), e.getY()));
        curves.add(curve);
    }
});
addMouseMotionListener(new MouseMotionAdapter() {
    @Override
    public void mouseDragged(MouseEvent e) {
        curves.get(curves.size() - 1)
              .add(new Point(e.getX(), e.getY()));
        repaint(0, 0, getWidth(), getHeight());
    }
});
```

Step	Line	Variables	Explanation
1	4: `var curve = new ArrayList<Point>();` 5: `curve.add(new Point(e.getX(), e.getY()));` 6: `curves.add(curve);`	e.getX()=40, e.getY()=120, curves.size()=1	**mousePressed**: Pressed at (40, 120): a new inner list is created, its first point is added, and it is put in the outer list.
2	12: `curves.get(curves.size() - 1)` 13: `.add(new Point(e.getX(), e.getY()));` 14: `repaint(0, 0, getWidth(), getHeight());`	e.getX()=60, e.getY()=60, curves.size()=1	**mouseDragged**: Dragged to (60, 60): a point is added to the last curve; `repaint` requests a redraw, and the line appears.
3	12: `curves.get(curves.size() - 1)` 13: `.add(new Point(e.getX(), e.getY()));` 14: `repaint(0, 0, getWidth(), getHeight());`	e.getX()=80, e.getY()=120, curves.size()=1	**mouseDragged**: Dragged to (80, 120): the same curve gets longer. In reality dozens of `mouseDragged` events arrive per second.
4	1: `addMouseListener(new MouseAdapter() {`	e.getX()=80, e.getY()=120, curves.size()=1	**mouseReleased**: Released: we did not write `mouseReleased`; the **empty body** of `MouseAdapter` runs and nothing happens.

Step	Line	Variables	Explanation
5	4: `var curve = new ArrayList<Point>();` 5: `curve.add(new Point(e.getX(), e.getY()));` 6: `curves.add(curve);`	e.getX()=50, e.getY()=95, curves.size()=2	**mousePressed**: Pressed again at (50, 95): the **second** curve starts. The old curve stays as it is.
6	12: `curves.get(curves.size() - 1)` 13: `.add(new Point(e.getX(), e.getY()));` 14: `repaint(0, 0, getWidth(), getHeight());`	e.getX()=70, e.getY()=95, curves.size()=2	**mouseDragged**: Dragged to (70, 95): the point is again added to the **last** curve, that is, curve 1.

Table 9.15: Trace of the mouse events.

In step 1 the outer list is empty. `mousePressed` creates a new inner list, adds `(40, 120)`, and puts the inner list in the outer list: `curves.size()` becomes 1. In steps 2 and 3, `mouseDragged` adds two points. Both go to the last curve, that is, curve 0. After each addition, `repaint` requests a redraw and the new line segment appears on the screen.

In real use the number of events is much larger. As the mouse moves, the operating system reports dozens of positions per second, and each becomes a `mouseDragged`. The two drags in the trace are two samples from this stream.

In step 4 the user releases the mouse. We did not write `mouseReleased`. The event still arrives, but the empty body of `MouseAdapter` runs and nothing happens. In step 5, a new press starts a second inner list; the old curve stays as it is. The drag in step 6 is again added to the last curve, now curve 1.

The event sequence was tried by sending it directly to the panel in headless mode. The data written after each step gave the final state of the table: `[[(40,120), (60,60), (80,120)], [(50,95), (70,95)]]`.

9.7.3 Code trace: drawing the curves

```
@Override
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    for (var c : curves) {
        var prev = c.get(0);
        for (var p : c) {
            g.drawLine(prev.x, prev.y, p.x, p.y);
            prev = p;
        }
    }
}
```

The method uses two nested loops. The outer loop goes over the curves and the inner loop over the points of one curve. The variable `prev` holds the point where the line will start. At the beginning of each curve, `prev` is set to the first point of that curve. At each point the inner loop draws a line from `prev` to that point and advances `prev`.

Step	Line	Variables	New output
1	3: `super.paintComponent(g);`	—	—
2	4: `for (var c : curves) {` 5: `var prev = c.get(0);`	c=curve 0, prev=(40,120)	—
3	6: `for (var p : c) {` 7: `g.drawLine(prev.x, prev.y, p.x, p.y);` 8: `prev = p;`	c=curve 0, prev=(40,120), p=(40,120)	`drawLine(40, 120, 40, 120)`
4	6: `for (var p : c) {` 7: `g.drawLine(prev.x, prev.y, p.x, p.y);` 8: `prev = p;`	c=curve 0, prev=(40,120), p=(60,60)	`drawLine(40, 120, 60, 60)`
5	6: `for (var p : c) {` 7: `g.drawLine(prev.x, prev.y, p.x, p.y);` 8: `prev = p;`	c=curve 0, prev=(60,60), p=(80,120)	`drawLine(60, 60, 80, 120)`
6	4: `for (var c : curves) {` 5: `var prev = c.get(0);`	c=curve 1, prev=(50,95)	—
7	6: `for (var p : c) {` 7: `g.drawLine(prev.x, prev.y, p.x, p.y);` 8: `prev = p;`	c=curve 1, prev=(50,95), p=(50,95)	`drawLine(50, 95, 50, 95)`
8	6: `for (var p : c) {` 7: `g.drawLine(prev.x, prev.y, p.x, p.y);` 8: `prev = p;`	c=curve 1, prev=(50,95), p=(70,95)	`drawLine(50, 95, 70, 95)`

Table 9.16: Trace of `paintComponent`; the "New output" column is the drawing call made.

One detail stands out in step 3. The inner loop starts at the first point, and `prev` is also the first point. The first call is `drawLine(40, 120, 40, 120)`: the point is joined to itself, and a single pixel is drawn. Then `prev = p` is executed. The next two calls draw the real lines: `(40,120) → (60,60)` and `(60,60) → (80,120)`.



Figure 9.24: Step 8. The drawing is complete after five `drawLine` calls; the two curves are not joined to each other.

In the next step the outer loop moves on to the second curve, and `prev` again becomes the first point of this curve. This line is what keeps the two curves from joining. If `prev` were not reset for each curve, an unwanted line would be drawn between the last point of curve 0, `(80, 120)`, and the first point of curve 1, `(50, 95)`. The total number of calls is 3 + 2 = 5; the number of visible line segments is 2 + 1 = 3.

The `DebugGraphics` log gave the same five calls:

```
setColor(238, 238, 238)
fillRect(0, 0, 200, 200)
drawLine(40, 120, 40, 120)
drawLine(40, 120, 60, 60)
drawLine(60, 60, 80, 120)
drawLine(50, 95, 50, 95)
drawLine(50, 95, 70, 95)
```

Again, `super.paintComponent(g)` produced the first two lines: the background was cleared with the panel's background color. No trace of the earlier drawing remains. The new picture is built each time from an empty background and the whole of the data.

Example 9.12

The user presses at `(10, 10)`, drags to `(20, 10)` and `(30, 20)`, and releases; then presses at `(5, 5)`, drags to `(6, 6)`, and releases. What are `curves.size()`, the number of points in each curve, and the number of `drawLine` calls when repainting?

Each press adds an inner list, so `curves.size()` is 2. Curve 0 has three points and curve 1 has two. The number of `drawLine` calls equals the number of points: 3 + 2 = 5. The log is as follows:

```
setColor(238, 238, 238)
fillRect(0, 0, 200, 200)
drawLine(10, 10, 10, 10)
drawLine(10, 10, 20, 10)
drawLine(20, 10, 30, 20)
drawLine(5, 5, 5, 5)
drawLine(5, 5, 6, 6)
```

9.7.4 `main` and `repaint`

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
        var frame = new JFrame("Simple Drawing Program");
        frame.getContentPane().add(new SimpleDrawPanel(),
                BorderLayout.CENTER);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);
        frame.setVisible(true);
    });
}
```

`main` follows the familiar pattern: the interface is built on the EDT with `invokeLater`, the panel is added to the center of the content pane, and the window is shown at 400 × 300.

The `repaint` method is used in two forms, and both are only requests:

- `repaint()` does not paint immediately; it puts a “repaint” request in the queue. Consecutive requests can be merged into a single painting. While the mouse produces dozens of events per second, this merging prevents unnecessary drawing.
- `repaint(x, y, w, h)` asks for only that rectangle to be repainted. Swing clips everything outside this region: `g` does not paint outside the region, and the work is reduced. `repaint(0, 0, getWidth(), getHeight())` is the same as `repaint()`.

Definition 9.15

`repaint()` is the method that leaves a request in the event queue for a component to be redrawn. At a suitable moment Swing calls the component's `paintComponent` method. The programmer does not call `paintComponent` directly; the programmer changes the data and requests `repaint()`.

Common mistake

Calling `paintComponent(getGraphics())` from inside a listener. This call bypasses Swing's painting machinery. Double buffering, that is, preparing the image in memory first, is switched off, and the picture may flicker. At the next real painting, everything drawn is erased as well. The correct way is `repaint()`.

9.8 Buttons and text components

9.8.1 The `JButton` class

`JButton` creates clickable buttons. Its text is set with `setText()` and its icon with `setIcon()`. `setEnabled(false)` disables the button: it looks faded and produces no event when clicked.

To respond to a click on a button, a listener is added with `addActionListener()`. The listener implements the `ActionListener` interface. This interface has a single method: `actionPerformed(ActionEvent e)`. A button can also be triggered with the space bar; the same method is called in both cases. That is why we listen for an “action” event, not a mouse event, on a button.

Adding a listener is an example of the Observer design pattern. The button is the “subject”; it keeps a list of the observers registered with it. When a click occurs, it notifies every observer in the list. The button does not know what the observers do. To add new behavior, you do not need to touch the button's code; you only register a new observer.

```
class CounterPanel extends JPanel {
    private int count = 0;
    private JLabel label = new JLabel("Not clicked yet");

    CounterPanel() {
        JButton button = new JButton("Click");
        button.addActionListener(e -> {
            count++;
            label.setText(count + " click(s)");
        });
        add(button);
        add(label);
    }
}
```

9.8.2 Code trace: the counter button

Step	Line	Call stack	Variables	Explanation
1	2: `private int count = 0;` 3: `private JLabel label = new JLabel("Not clicked yet");`	main() → new CounterPanel()	count=0, label=JLabel	When the object is created, the fields get their initial values first: `count` is 0 and the label is ready.
2	6: `JButton button = new JButton("Click");`	main() → CounterPanel()	button=JButton("Click")	The constructor creates a button; `button` is a local variable.
3	7: `button.addActionListener(e -> {` 8: `count++;` 9: `label.setText(count + " click(s)");` 10: `});`	main() → CounterPanel()	listener count=1	The lambda is **not run**; it is only registered with the button. Its body will run on a click.
4	11: `add(button);` 12: `add(label);`	main()	component count=2	The button and the label are added to the panel (`FlowLayout`: side by side). The constructor ends.
5	8: `count++;`	EDT → actionPerformed(e)	count=1	The user clicks: the EDT calls `actionPerformed`, that is, the lambda body. `count` can be incremented because it is a **field**.
6	9: `label.setText(count + " click(s)");`	EDT → actionPerformed(e)	count=1	`setText` changes the text of the label; Swing repaints the label.
7	8: `count++;`	EDT → actionPerformed(e)	count=2	The second click: the same lambda is called again, and the field becomes 2.

Step	Line	Call stack	Variables	Explanation
8	9: `label.setText(count + " click(s)");`	EDT → actionPerformed(e)	count=2	The label is updated. Between clicks the program runs no code at all; it only **waits for events**.

Table 9.17: Trace of the counter button.

In step 1, `new CounterPanel()` starts to run. Before the body of the constructor, the fields get their initial values: `count` becomes 0 and `label` is created with the text “Not clicked yet.” In step 2, the constructor creates a button. `button` is a local variable; it disappears when the constructor ends, but the button itself lives on as a component of the panel.

Step 3 is the most important step of the trace. The lambda expression is not run at this point. It is only registered with the button as an object. `count++` and `setText` in its body will run only when the button is clicked. In step 4, the button and the label are added to the panel. `FlowLayout` places them side by side. The constructor ends; the panel has 2 components and the button has 1 listener.

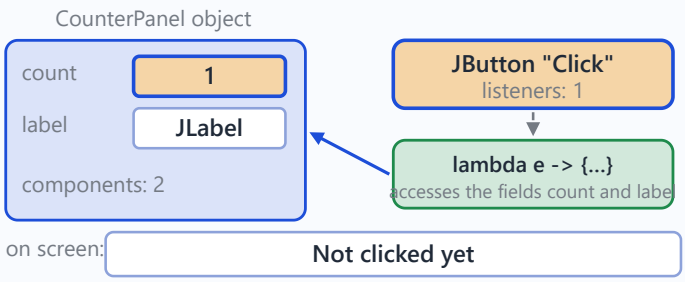


Figure 9.25: Step 5. The user has clicked; the EDT called the `actionPerformed` of the listener registered with the button, that is, the lambda body.

In step 5 the user clicks. The EDT calls `actionPerformed`, that is, the lambda body. `count` can be incremented because it is a field: it becomes 1. In step 6, `setText` changes the label's text to “1 click(s)” and Swing repaints the label. Steps 7 and 8 are the second click: the same lambda is called again, the field becomes 2, and the label is updated.

Between clicks the program runs no code; it only waits for events. This behavior is very different from the continuously running flow of a console program. When a panel was created in headless mode and two clicks were given with the button's `doClick` method, the label became “1 click(s)” and “2 click(s)”. When the button was disabled with `setEnabled(false)` and clicked again, the label did not change.

```
2 1 Not clicked yet
1 click(s)
2 click(s)
disabled: 2 click(s)
```

9.8.3 Anonymous classes and lambdas

The same listener can be written in two forms. The classic form is an anonymous inner class:

```
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        count++;
        label.setText("Count: " + count);
    }
});
```

Since Java 8, the same job can be written with a lambda expression:

```
button.addActionListener(e -> {
    count++;
    label.setText("Count: " + count);
});

// one expression: no braces needed
button.addActionListener(e -> System.exit(0));
```

A lambda can be used because of the structure of `ActionListener`: the interface has only one abstract method.

Definition 9.16

A **functional interface** is an interface with a single abstract method. A **lambda expression** is taken as the body of the single method of such an interface: `e -> …` is the body of `actionPerformed(e)`.

The compiler looks at the parameter type of `addActionListener`, sees `ActionListener`, and finds its single method. The lambda's parameter `e` becomes the `ActionEvent` parameter of that method. If the body is a single expression, the braces are not needed either: `button.addActionListener(e -> System.exit(0));`.

Since `MouseListener` has five methods, it cannot be written as a lambda. The compiler cannot tell which method's body is being written:

```
Main.java:9: error: incompatible types: MouseListener is not a functional interface
  p.addMouseListener(e -> System.out.println("click"));
    multiple non-overriding abstract methods found in interface MouseListener
```

That is why an adapter class is used for mouse events.

A lambda has one more limitation: a local variable cannot be modified from inside it. A lambda captures (copies) the value of the local variable it uses; it is not bound to the variable itself. For this reason a local variable must be final or “effectively final,” that is, never changed after it is defined.

Common mistake

Keeping the counter in a local variable: `int count = 0; … b.addActionListener(e -> count++);` . The compiler gives this message:

```
Main.java:10: error: local variables referenced from a lambda expression must be
final or effectively final
  b.addActionListener(e -> count++);
```

Changing values such as a counter must be fields of the class. Fields belong to the object; the lambda holds a reference to the object and can change the field. The counter panel therefore uses the field `private int count` .

9.8.4 `JLabel` and `JTextField`

`JLabel` is a label. It displays text or an image; the user cannot change it. `setText()` sets the text, `setIcon()` the image, `setFont()` the font, and `setForeground()` the text color.

`JTextField` is a single-line text field. It is for the user to enter text; data from forms is collected this way. `getText()` and `setText()` read and write the field. `setColumns()` sets the number of visible columns. This number affects the preferred width of the field; it does not limit the length of text it can hold. In headless mode, a 12-column field asked for a width of 136 pixels and a 24-column field for 268. A listener added with `addActionListener()` runs when the user presses Enter in the field.

```
JLabel label = new JLabel("Your name:");
label.setFont(new Font("SansSerif", Font.BOLD, 14));
label.setForeground(Color.BLUE);
JTextField nameField = new JTextField(12);
JLabel greeting = new JLabel(" ");
nameField.addActionListener(e -> greeting.setText(
        "Hello, " + nameField.getText() + "!"));
```

The code creates two labels and a text field. The first label is bold, 14-point, and blue. The text field is 12 columns wide. The second label initially contains a space: the preferred size of a label with empty text is 0 × 0, whereas a label with a single space takes up 3 × 16 pixels and reserves the greeting's row

from the start. The listener reads the text in the field when Enter is pressed and writes it to the greeting label. In headless mode, when "Ayşe" was typed into the field and an Enter event was generated, the greeting label became `Hello, Ayşe!`.

Inside the lambda, the local variables `nameField` and `greeting` are used. Because these variables are not modified after they are defined, they are effectively final, and the lambda can capture them. The contents of the object a variable refers to may change; `greeting.setText(…)` is an example. The rule only forbids redirecting the variable to another object.

9.8.5 `JTextArea` and `JScrollPane`

`JTextArea` is a multi-line text area. Its content can be read, assigned, and updated. The visible size is given in advance with `new JTextArea(rows, columns)`. Its commonly used methods are: `getText()`, `setText()`, `append()` (adds at the end), `setEditable(false)` (makes it read-only), and `setLineWrap(true)` (wraps long lines).

`JScrollPane` is a scroll pane. When the component placed in it (a text area, a table, an image) does not fit on the screen, it adds scroll bars. It adapts itself when the size of the content changes. Its own size is set with `setPreferredSize()` and its content with `setViewportView()`. The text area is given to the scroll pane in the constructor, and it is the scroll pane, not the text area, that is added to the window.

```
JTextArea area = new JTextArea(5, 20);
area.setText("First line\n");
area.append("Second line\n");
JScrollPane scroll = new JScrollPane(area);
System.out.print(area.getText());
System.out.println(area.getLineCount());
```

```
First line
Second line
3
```

`setText` deletes the old content and writes the new; `append` adds at the end. The text becomes `"First line\nSecond line\n"`. `getLineCount()` returns one more than the number of line breaks: the text has two line breaks, so the result is 3. The third line is empty, but it is counted because the caret can be placed there. Even in an empty text area, `getLineCount()` is 1.

Example 9.13

Which three lines does the following code print?

```
JTextArea a = new JTextArea("x");
a.append("\ny");
a.append("\n");
System.out.println(a.getLineCount());
a.setText("one");
a.append(" line");
System.out.println(a.getText() + " " + a.getLineCount());
JTextField f = new JTextField(10);
f.setText(" 42 ");
System.out.println(Integer.parseInt(f.getText().trim()) + 1);
```

The text becomes `"x\ny\n"` : two line breaks, three lines. The first line is `3` . `setText` deletes the old content and `append` adds: `"one line"` , with no line break, so the line count is 1. `getText()` always returns a `String` . `trim()` removes the leading and trailing spaces; `42 + 1 = 43` . If the spaces had not been removed, `parseInt` would have thrown this exception: `NumberFormatException: For input string: " 42 "` .

```
3
one line 1
43
```

9.9 Application: a text converter

This application converts the text in a text area to lowercase or uppercase with two buttons. The layout of the window is a good example of using two layout managers together:

- In the center (`CENTER`) there is a `JTextArea` of 8 rows × 50 columns inside a `JScrollPane` .
- At the bottom (`SOUTH`) there is a `JPanel` holding two buttons. Since the panel uses `FlowLayout` , the buttons are centered and side by side.
- `pack()` computes the window size from the preferred sizes of these components.
- The buttons convert the text with `toLowerCase()` or `toUpperCase()` and write it back to the area.

```
import java.awt.BorderLayout;
import javax.swing.*;

public class CaseConverter {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            var area = new JTextArea("Java Programming", 8, 50);
            var lowerButton = new JButton("To Lowercase");
            var upperButton = new JButton("To Uppercase");
            lowerButton.addActionListener(e ->
                    area.setText(area.getText().toLowerCase()));
            upperButton.addActionListener(e ->
                    area.setText(area.getText().toUpperCase()));
            var panel = new JPanel();
            panel.add(lowerButton);
            panel.add(upperButton);
            var frame = new JFrame("CaseConverter");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.getContentPane().add(new JScrollPane(area),
                    BorderLayout.CENTER);
            frame.getContentPane().add(panel, BorderLayout.SOUTH);
            frame.pack();
            frame.setVisible(true);
        });
    }
}
```

The whole interface is built inside a single lambda, on the EDT. First the text area and the two buttons are created. A listener is added to each button. The listener reads the text of the area, converts it, and writes the result back to the area. Then the buttons are put in a panel. The window is created, the scroll pane is added to the center and the button panel to the south. `pack()` computes the size, and the window is shown.

The code has three details:

- `area` is a local variable, but it can be used inside the lambda. After it is defined, the variable is never redirected to another object; it is effectively final. The lambda uses not the variable `area` itself but the object it refers to, and it changes the text of that object.
- `toUpperCase()` returns a new `String`. Since `String` is immutable, the original text does not change; the result must be written back to the area with `setText`. If `area.getText().toUpperCase();` were written alone, nothing on the screen would change.
- `BorderLayout.CENTER` is written with its full name. Writing only `CENTER` does not compile, because the constant is inside the `BorderLayout` class.

When the same two buttons and a text area were created in headless mode and the buttons were pressed with `doClick`, the text turned first to uppercase and then to lowercase:

```
JAVA PROGRAMMING
java programming
default locale tr_TR: İSTANBUL / ENGLISH ISTANBUL / tr İSTANBUL
```

9.9.1 What happens on a click?

Between the user's click on the “To Uppercase” button and the text changing on the screen, there are six steps.

1. The user clicks the button; the operating system passes this to Java.
2. Java creates an event object and puts it in the event queue.
3. The EDT takes the next event. The button calls `actionPerformed` of all registered `ActionListener`s: our lambda runs.
4. `area.getText()` reads the current text.
5. `toUpperCase()` produces a new string; the old string does not change.
6. `setText` updates the model of the area. The area has itself repainted with `repaint`. The EDT moves on to the next event.

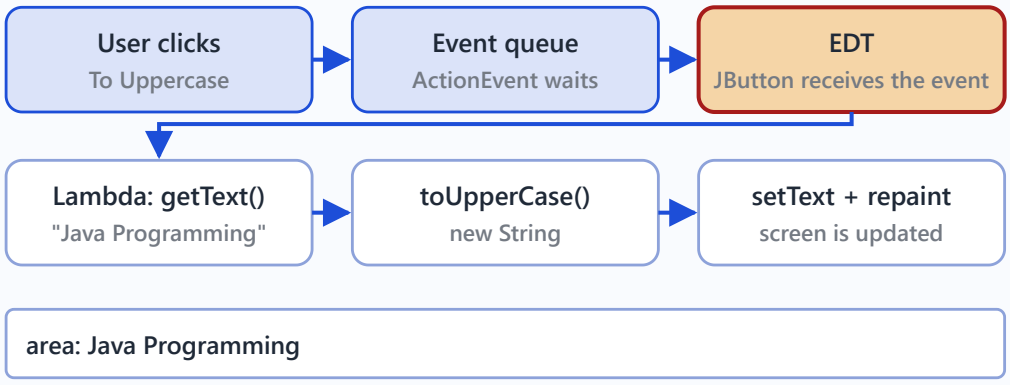


Figure 9.26: Step 3. The EDT took the event from the queue and called the listener registered with the button.

Step 6 mentions the word “model.” In Swing, text components consist of two parts: a model that holds the text and a view that displays it on the screen. `setText` changes the model; the view notices the change and has itself redrawn. The change calculator application uses this separation directly.

9.9.2 Uppercase and locale

When `toUpperCase()` and `toLowerCase()` are called without arguments, they use the computer's locale. Turkish is one of the languages with special rules for letter case conversion: there are two separate letters “i,” dotted and dotless. On a machine with English settings,

`"istanbul".toUpperCase()` gives `"ISTANBUL"`; with Turkish settings it gives `"İSTANBUL"`. The same program gives different results depending on the computer it runs on.

To fix the result, the locale is given explicitly: `toUpperCase(Locale.forLanguageTag("tr"))`. The following experiment compares the rules of the two languages.

```
String s = "Kış İzmir";
Locale tr = Locale.forLanguageTag("tr");
System.out.println(s.toUpperCase(tr));
System.out.println(s.toLowerCase(tr));
System.out.println(s.toUpperCase(Locale.ENGLISH));
System.out.println(s);
```

```
KIŞ İZMİR
kış izmir
KIŞ İZMIR
Kış İzmir
```

In Turkish rules, `i → İ`, `ı → I`, `İ → i`, and `I → ı`. In English rules, both `ı` and `i` become `I` in uppercase, while `İ` stays as it is. That is why the third line is `KIŞ İZMIR`. The last line shows once more that strings are immutable: after the three conversions, `s` is still `"Kış İzmir"`.

A locale object used to be created with `new Locale("tr")`. This constructor has been deprecated since Java 19, and the compiler gives a warning. Use `Locale.forLanguageTag("tr")` or `Locale.of("tr")` instead.

Common mistake

Basing the logic of a program on a locale-dependent conversion. For example, if a command is checked with `command.toUpperCase().equals("QUIT")`, on a machine with Turkish settings `"quit"` becomes `"QUİT"` and the comparison fails. For internal strings that are not shown to the user, give `Locale.ROOT` or `Locale.ENGLISH`; for Turkish text shown to the user, give the Turkish locale.

9.10 Application: a change calculator

This application shows how an entered number of cents can be paid with the fewest coins (50, 25, 10, 5, and 1 cent). The window has no button: the report is updated automatically on every keystroke. This is provided by a `DocumentListener` added to the document of the text field.

9.10.1 Fields and constructor

```
public class ChangeCalculator extends JFrame {
    private JTextField amountField = new JTextField(12);
    private Document doc = amountField.getDocument();
    private JTextArea report = new JTextArea(8, 40);

    public ChangeCalculator() {
        var topPanel = new JPanel();
        topPanel.add(new JLabel("Amount:"));
        topPanel.add(amountField);
        getContentPane().add(topPanel, BorderLayout.NORTH);
        getContentPane().add(new JScrollPane(report),
                BorderLayout.CENTER);
        setBackground(Color.LIGHT_GRAY);
        report.setEditable(false);
        doc.addDocumentListener(new DocumentListener() {
            public void insertUpdate(DocumentEvent e) { updateReport(); }
            public void removeUpdate(DocumentEvent e) { updateReport(); }
            public void changedUpdate(DocumentEvent e) { updateReport(); }
        });
    }
```

This time the class derives not from `JPanel` but from `JFrame`. The window itself is the application. There are three fields: an amount field of 12 columns, the document of this field, and a report area of 8 rows × 40 columns.

In Swing, data and view are separate. This arrangement is called Model-View-Controller (MVC). `JTextField` is the view: it shows the text on the screen and receives what the user types. The `Document` object obtained with `getDocument()` is the model that holds the text. When the user presses a key, the model changes and everyone listening to the model is notified.

The constructor builds the top panel: an “Amount:” label and the amount field. The panel is added to the north of the window and the report area, inside a scroll pane, to the center. The report is made read-only. Finally an anonymous `DocumentListener` is added to the document.

`DocumentListener` has three methods: `insertUpdate` (text was inserted), `removeUpdate` (text was removed), and `changedUpdate` (style changed). In a plain text field there is no style, so `changedUpdate` is never called. But all three methods of the interface are abstract, and all three must be written. All three do the same thing: they call `updateReport()`.

9.10.2 Code trace: `updateReport`

```
void updateReport() {
    try {
        var amount = Integer.parseInt(
                doc.getText(0, doc.getLength()));
        int[] coins = {50, 25, 10, 5, 1};
        report.setText(amount
                + " cents in coins:\n");
        for (int x : coins) {
            report.append(amount / x + " x " + x + "c\n");
            amount %= x;
        }
    } catch (NumberFormatException e) {
        report.setText("Not an integer or out of range");
    } catch (Exception e) {
        report.setText(e.toString());
    }
}
```

The method reads the text of the document and converts it to an integer. `doc.getText(0, doc.getLength())` returns the text of the document starting at character 0 and extending over its whole length. Then the coin values are put in an array from largest to smallest. The report is restarted from scratch with `setText`. For each coin value, the loop does two things: it prints how many of that coin will be used (`amount / x`) and computes the remaining amount (`amount %= x`).

Step	Line	Variables	New output
1	3: `var amount = Integer.parseInt(` 4: `doc.getText(0, doc.getLength()));`	amount=93	—
2	5: `int[] coins = {50, 25, 10, 5, 1};`	amount=93, coins=[50, 25, 10, 5, 1]	—
3	6: `report.setText(amount` 7: `+ " cents in coins:\n");`	amount=93, coins=[50, 25, 10, 5, 1]	`93 cents in coins:`
4	9: `report.append(amount / x + " x " + x + "c\n");`	amount=93, x=50	`1 x 50c`
5	10: `amount %= x;`	amount=43, x=50	—
6	9: `report.append(amount / x + " x " + x + "c\n");`	amount=43, x=25	`1 x 25c`
7	10: `amount %= x;`	amount=18, x=25	—
8	9: `report.append(amount / x + " x " + x + "c\n");`	amount=18, x=10	`1 x 10c`
9	10: `amount %= x;`	amount=8, x=10	—
10	9: `report.append(amount / x + " x " + x + "c\n");`	amount=8, x=5	`1 x 5c`

Step	Line	Variables	New output
11	10: `amount %= x;`	amount=3, x=5	—
12	9: `report.append(amount / x + " x " + x + "c\n");`	amount=3, x=1	`3 x 1c`
13	10: `amount %= x;`	amount=0, x=1	—

Table 9.18: Trace of `updateReport` for the amount 93.

The method is a greedy algorithm: at each step the largest coin that can be used is chosen. For 93 cents, first a 50 is used, leaving 43. Then a 25, leaving 18. A 10, leaving 8. A 5, leaving 3. Finally three 1s. Seven coins in all. Integer division gives the count and the remainder operator gives the remaining amount.



Figure 9.27: Step 13. The loop has finished: 1 × 50, 1 × 25, 1 × 10, 1 × 5, and 3 × 1 cents.

For the coins 50, 25, 10, 5, 1, the greedy method always gives the fewest coins. This is not true for every coin system. For example, if the coins were 1, 3, and 4, then for 6 the greedy method would choose 4 + 1 + 1 (three coins), whereas 3 + 3 gives the same amount with two coins.

Example 9.14

The user typed 187 into the amount field. Which lines does the report contain?

187 / 50 = 3, remainder 37. 37 / 25 = 1, remainder 12. 12 / 10 = 1, remainder 2. 2 / 5 = 0, remainder 2. 2 / 1 = 2, remainder 0. In total 3 + 1 + 1 + 0 + 2 = 7 coins. The line with a count of zero is also written, because the loop adds a line for every coin value:

```
187 cents in coins:
3 x 50c
1 x 25c
1 x 10c
0 x 5c
2 x 1c
```

9.10.3 `main` and the order of events

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
        var frame = new ChangeCalculator();
        frame.setTitle("Change Calculator");
        frame.pack();
        frame.setDefaultCloseOperation(
                JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    });
}
```

`main` creates the window on the EDT, gives it a title, computes its size with `pack()`, and shows it. The really interesting part of the program is which events arrive depending on what the user types. Since a `JFrame` cannot be created in headless mode, the same fields, the listener, and the `updateReport` method were tried in a class without a window; insertions into and removals from the document were done directly. The events that arrived are these:

```
types 9                 insert
types 3                 insert
deletes everything      remove
abc                     insert
99999999999             remove insert
setText 287             remove insert
setText empty           remove
setText 287 (was empty) insert
-7                      remove insert
68                      remove insert
187                     remove insert
with spaces             remove insert
```

User / code	Event	Report
Types 9	`insertUpdate`	9 cents: 0, 0, 0, 1 × 5, 4 × 1
Types 3 (field is 93)	`insertUpdate`	93 cents: 1, 1, 1, 1, 3
Deletes everything	`removeUpdate`	`parseInt("")` : “Not an integer or out of range”
`abc` or `99999999999`	`NumberFormatException`	The same message
`setText("287")` (while the field has text)	First `removeUpdate`, then `insertUpdate`	5, 1, 1, 0, 2
`setText("287")` (while the field is empty)	Only `insertUpdate`	5, 1, 1, 0, 2
`-7`	A valid number	0, 0, 0, −1, −2: a check is missing

Table 9.19: User actions, events, and the report (verified by running).

Every row of the table carries a lesson. While the user types “93,” two events arrive: one `insertUpdate` each for “9” and “3.” The report is updated on every key; the intermediate state (9 cents) is also visible for a moment. When the field is completely cleared, the text is the empty string and `parseInt("")` throws `NumberFormatException`. The first `catch` catches it.

`99999999999` is a number, but it exceeds 2,147,483,647, the largest value of `int`. In this case too, `parseInt` throws `NumberFormatException`; that is why the message says “not an integer *or out of range*.” `setText` works not as a single operation but in two steps: it first removes the old text and then inserts the new one. If the field has text, two events arrive. At the first event the field is momentarily empty and the report shows the error message; the second event writes the correct report.

A negative number reveals a deficiency. `-7` is a valid integer. In Java, integer division truncates toward zero and the remainder takes the sign of the dividend: `-7 / 5 = -1`, `-7 % 5 = -2`. The report contains meaningless lines such as `-1 x 5c` and `-2 x 1c`:

```
-7 cents in coins:
0 x 50c
0 x 25c
0 x 10c
-1 x 5c
-2 x 1c
```

The program should reject the negative value. A check right after `parseInt` is enough: `if (amount < 0) { report.setText("Amount must be positive"); return; }`.

The two `catch` blocks have different jobs. The first catches the expected error, that is, text that is not a number and values beyond the `int` limit. The second is a general safeguard for everything else. This block is necessary because `getText(0, n)` declares that it may throw `BadLocationException`, a checked exception. The compiler requires this exception to be handled; `catch (Exception e)` covers it too.

Example 9.15

What does the report show when 68 is typed into the change program?

68 / 50 = 1, remainder 18. 18 / 25 = 0, remainder 18. 18 / 10 = 1, remainder 8. 8 / 5 = 1, remainder 3. 3 / 1 = 3. The report begins with the line “68 cents in coins:” and continues with five lines:

```
68 cents in coins:
1 x 50c
0 x 25c
1 x 10c
1 x 5c
3 x 1c
```

9.11 Best practices

A few habits determine the quality of GUI programs.

Use a layout manager
: Do not give coordinates by hand. With a layout manager, the layout does not break at a different screen size, resolution, or font. Complex layouts are built with nested panels.

Choose meaningful names
: Use names such as `upperButton` and `amountField` instead of `b1` and `t2`. In an interface with dozens of components, the code stays readable only this way.

Think in terms of events
: Design starts from the question “What should happen when the user does this?” A listener is written for each action; listeners do not rely on one another's order.

Separate the interface from the business logic
: Rules such as the change calculation are kept in a separate method or class. That way they can be tested without the interface and used in another interface. Most of the outputs given in this chapter were obtained without opening a window, thanks to this separation.

Build the interface on the EDT
: The body of `main` is a single call: `SwingUtilities.invokeLater(() -> { … })`.

Do not keep the EDT busy
: If a listener does a long job (downloading a file, a large computation), the EDT cannot process any other event for that time and the window freezes. Such jobs are moved to a separate thread; the result is handed back to the EDT with `invokeLater`.

Example 9.16

How can the change calculation be separated from the interface?

The calculation is moved to a static method that takes an amount as input and returns the counts:

```
static int[] makeChange(int amount) {
    int[] coins = {50, 25, 10, 5, 1};
    int[] counts = new int[coins.length];
    for (int i = 0; i < coins.length; i++) {
        counts[i] = amount / coins[i];
        amount %= coins[i];
    }
    return counts;
}
```

`updateReport` now only reads the text, calls this method, and writes the result to the report. The method itself can be tried directly in a console program or a test: `makeChange(93)` returns `[1, 1, 1, 1, 3]`.

9.12 Solved problems

Example 9.17 (Problem 1)

If a button `A` and then a button `B` are added to the same `NORTH` region of a panel that uses `BorderLayout`, which one is visible in the north?

`B`. Each region holds a single component; the last one added wins. `A` remains in the panel (the component count is 2) but takes no part in the layout. In headless mode, on a 200 × 100 panel, `B` came out as `(0, 0, 200, 26)` and `A` as 0 × 0.

Example 9.18 (Problem 2)

If the user clicks the panel once and releases, which methods of `MouseListener` are called, and in what order?

`mousePressed` → `mouseReleased` → `mouseClicked`. If the pointer enters the panel from outside, `mouseEntered` arrives as well. If the mouse moves between the press and the release, `mouseDragged` events come in between and `mouseClicked` may not arrive.

Example 9.19 (Problem 3)

Which color does `Color.RED.brighter()` give?

Again `(255, 0, 0)`. The red component is already at the maximum; 255 / 0.7 = 364 is capped at 255. The green and blue components are 0, and 0 / 0.7 = 0 stays.

Example 9.20 (Problem 4: a target)

A target made of five nested red and white rings, scaled to the size of the panel, is to be drawn. How is `paintComponent` written? Which calls are made on a 300 × 200 panel?

The center and the largest radius are computed as in the sign. In each round the loop reduces the radius by one fifth and alternates the colors:

```
int cx = getWidth() / 2, cy = getHeight() / 2;
int r = Math.min(cx, cy) - 5;
for (int i = 0; i < 5; i++) {
    int ri = r - i * r / 5;
    g.setColor(i % 2 == 0 ? Color.RED : Color.WHITE);
    g.fillOval(cx - ri, cy - ri, 2 * ri, 2 * ri);
}
```

On 300 × 200, `cx = 150`, `cy = 100`, and `r = 95`. `i * r / 5` is evaluated left to right: first the multiplication, then the integer division. The radii are 95, 76, 57, 38, and 19; the calls are `fillOval(55, 5, 190, 190)`, `(74, 24, 152, 152)`, `(93, 43, 114, 114)`, `(112, 62, 76, 76)`, and `(131, 81, 38, 38)`. The circles are drawn from largest to smallest. If they were drawn from smallest to largest, the last, large circle would cover all the others.

Example 9.21 (Problem 5: undo with the right button)

In the drawing program, let a right click delete the last curve, and let dragging with the right button not draw anything. Which changes are needed?

A check is added at the beginning of `mousePressed`. If it is the right button, the last curve is removed and the method returns:

```
if (e.getButton() == MouseEvent.BUTTON3) {
    if (!curves.isEmpty()) curves.remove(curves.size() - 1);
    repaint();
    return;
}
```

At the beginning of `mouseDragged`, write `if (!SwingUtilities.isLeftMouseButton(e)) return;` as well. Without this line, dragging with the right button would add points to the previous curve in place of the one just deleted. When the list is empty, a right drag would throw `IndexOutOfBoundsException: Index -1 out of bounds for length 0` through `get(-1)`. The changes were tried by sending events in headless mode: after two curves were drawn, a right click reduced the number of curves to 1, and a right drag changed no points.

Example 9.22 (Problem 6: a temperature converter)

When °C is entered in a `JTextField` and a button is pressed, °F should be shown. How is the listener written?

```
convert.addActionListener(e -> {
    try {
        double c = Double.parseDouble(field.getText());
        result.setText(c * 9 / 5 + 32 + " °F");
    } catch (NumberFormatException ex) {
        result.setText("Not a number");
    }
});
```

When 100 is entered, `212.0 °F` is shown; when −40 is entered, `-40.0 °F` is shown; −40 is the only temperature at which the two scales are equal. When `abc` is entered, `parseDouble` throws `NumberFormatException` and "Not a number" appears. In the expression `c * 9 / 5 + 32 + " °F"` the order matters: the numeric operations are done first, from left to right, and the string is appended last. The exception parameter inside the lambda is named `ex` instead of `e`, because `e` is already the lambda's parameter.

Two smaller exercises use the same patterns. In a counter window, the buttons "−1," "+1," and "Reset" change a `private int count` field and update the label. After two clicks on "+1" and one on "−1," the label shows `1`; after "Reset," it shows `0`. To add a "Clear" button to the drawing program, the button's listener runs `curves.clear(); repaint();`. The panel and the button go in the `CENTER` and `SOUTH` regions of the window.

9.13 Common mistakes

In graphics programs, some mistakes come from the compiler, while others appear as the program silently misbehaving. The compiler messages were obtained with JDK 21.

Mistake	Symptom	Fix
Forgetting the import	`cannot find symbol … class JFrame`	`import javax.swing.*;`
No `setVisible(true)`	The window never appears	`setVisible(true)` at the end of the setup
No `EXIT_ON_CLOSE`	The window closes, but the program keeps running in the background	`setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE)`
Misspelling the method name (`paintcomponent`)	It is never called; with `@Override` : `method does not override or implement a method from a supertype`	The correct name, and always `@Override`

Mistake	Symptom	Fix
Writing `paintComponent` without `public` / `protected`	`attempting to assign weaker access privileges; was protected`	`protected` or `public`
Forgetting `super.paintComponent(g)`	The background is not cleared, and remnants of old drawings stay	The first line of the body
Not writing all the methods of the interface	`… is not abstract and does not override abstract method mouseExited(MouseEvent) in MouseListener`	Derive from `MouseAdapter`
Not registering the motion listener	`mouseDragged` is never called	`addMouseMotionListener(…)`
`get(size())` for the last element	`IndexOutOfBoundsException: Index 1 out of bounds for length 1`	`get(size() - 1)`
Modifying a local variable inside a lambda	`local variables referenced from a lambda expression must be final or effectively final`	Make the variable a field
Passing a lambda for an interface with several methods	`MouseListener is not a functional interface`	An adapter or an anonymous class
Writing the constant without the class name	`cannot find symbol … variable CENTER`	`BorderLayout.CENTER`
`List` with `java.awt.*` and `java.util.*`	`reference to List is ambiguous`	`import java.util.List;`
`invokeAndWait` from inside the EDT	`java.lang.Error: Cannot call invokeAndWait from the event dispatcher thread`	Call directly on the EDT, or use `invokeLater`
A long job in a listener	The window freezes and does not respond to clicks	Move the job to a separate thread

Table 9.20: Common mistakes, their symptoms, and fixes.

When the import is forgotten, the compiler gives two errors for the same line: one for the type of the variable and one for the `new` expression.

```
Main.java:3: error: cannot find symbol
    JFrame f = new JFrame("x");
  symbol:   class JFrame
  location: class Main
Main.java:3: error: cannot find symbol
    JFrame f = new JFrame("x");
  symbol:   class JFrame
  location: class Main
```

The value of the `@Override` annotation is seen most clearly in a misspelled method name. Without the annotation, a class that writes `paintcomponent` (lowercase c) compiles without trouble. A new method has been defined, and Swing never calls it; the panel stays empty. When the annotation is added, the compiler finds the error immediately:

```
Main.java:7: error: method does not override or implement a method from a supertype
  @Override
```

`paintComponent` is defined as `protected` in `JComponent`. An overriding method cannot narrow the access: it can be `protected` or `public`. If no access modifier is written, the method gets package access and the compiler rejects it:

```
Main.java:7: error: paintComponent(Graphics) in Main cannot override
paintComponent(Graphics) in JComponent
  void paintComponent(Graphics g) {
  attempting to assign weaker access privileges; was protected
```

Writing the constant without the class name gives a similar message:

```
Main.java:9: error: cannot find symbol
  p.add(new JButton("x"), CENTER);
  symbol:   variable CENTER
  location: class Main
```

Example 9.23

A drawing panel compiles, but the curves are not visible, remnants of old drawings stay, and an exception is thrown on the first drag. The code contains these lines: `curves.get(curves.size()).add(e.getPoint())` and `public void paintcomponent(Graphics g) { for (…) … }`. What are the three mistakes?

The first is the name of the method: `paintcomponent` is a new method, and Swing does not call it. If `@Override` had been added, there would have been a compile error. Second, there is no `super.paintComponent(g)` inside the method; once the name is corrected, the background is not cleared and remnants stay. Third is the index of the last element: `get(size())` throws `IndexOutOfBoundsException`; the correct form is `size() - 1`. That the code compiles does not show that it is free of errors; none of the three mistakes is caught by the compiler.

9.14 Summary

- A GUI program is event-driven: `main` only builds the interface, and the user's actions determine the flow. Most of the code consists of listeners.
- The `java.awt` package contains `Graphics`, `Color`, `Font`, `Point`, the layout managers, and the event classes; the `javax.swing` package contains the components whose names start with `J`. Java itself draws the Swing components.

- In screen coordinates, `(0, 0)` is the top-left corner of the component; `x` grows to the right and `y` downward.
- For permanent drawing, derive from `JPanel` and override `paintComponent(Graphics g)`. The first line of the body is `super.paintComponent(g)`. Swing calls the method.
- The `draw…` methods draw outlines and the `fill…` methods draw filled shapes. Ellipses and circles are given by their bounding rectangle: a circle with center `(cx, cy)` and radius `r` is `fillOval(cx – r, cy – r, 2r, 2r)`. The `y` of `drawString` is the baseline.
- A `Graphics` object remembers its color and font; `setColor` and `setFont` affect all later drawing. A shape drawn later covers an earlier one.
- `JFrame` is a top-level window. The size is given with `setSize` or `pack()`, the close behavior is set with `setDefaultCloseOperation(EXIT_ON_CLOSE)`, and `setVisible(true)` is called last. The default close behavior is `HIDE_ON_CLOSE`.
- `JPanel` is both a container and a canvas; its default layout manager is `FlowLayout`. The content pane of a `JFrame` uses `BorderLayout`.
- The fields `x` and `y` of `Point` are `public int`; `getX()` and `getY()` return `double`. `translate` moves relatively and `setLocation` moves absolutely.
- `Color` is created from an RGB triple (0–255); it is immutable. `darker()` multiplies the components by 0.7 and `brighter()` divides them by 0.7; both truncate the result to integers and are not exact inverses of each other.
- `Font(name, style, size)`; the style constants are bit flags: `BOLD | ITALIC = 3`. `deriveFont(float)` changes the size and `deriveFont(int)` changes the style.
- A layout manager computes the position and size of components by a rule. `add` only appends to the list; placement happens when the layout is computed.
- `FlowLayout` arranges components row by row, centered, with a 5-pixel gap. `BorderLayout` places them in five regions: one component per region, the last one added wins, and the center takes the rest. `GridLayout` divides the container into equal cells. `BoxLayout` builds a single row or a single column.
- A listener interface defines the methods for an event; a listener is registered with `add…Listener`. `MouseListener` reports press, release, click, enter, and exit; `MouseMotionListener` reports movement and dragging. Registration is by interface.
- An adapter class (`MouseAdapter`) implements all the methods with empty bodies; a subclass writes only what it needs. An anonymous inner class suits a single-use listener.
- A click produces `pressed → released → clicked`. `MouseEvent` gives the position relative to the component; `getButton()` tells the button (`BUTTON3` is the right one).
- Swing is single-threaded: components are built and modified only on the EDT. `invokeLater` puts a task in the queue and returns; `invokeAndWait` waits until the task finishes and throws an `Error` if called from inside the EDT. The queue is ordered.
- A listener does not draw on the screen; it updates the data and requests `repaint()`. `repaint()` is a request; Swing calls `paintComponent` at a suitable moment.

- `ActionListener` is a functional interface with a single method and can be written as a lambda. A local variable cannot be modified from inside a lambda; values that change become fields.
- `JLabel` shows text, `JTextField` takes a single line and `JTextArea` multiple lines of text, and `JScrollPane` scrolls. `getLineCount()` is one more than the number of line breaks.
- `toUpperCase()` depends on the locale; in Turkish, `i → İ`. To fix the result, give the locale explicitly.
- In text components the model (`Document`) and the view are separate. A `DocumentListener` is called on every insertion and removal; `setText` produces two events on a non-empty field.
- Best practices: use a layout manager, choose meaningful names, separate the business logic from the interface, build the interface on the EDT, and do not keep the EDT busy with long jobs.

9.15 Key terms

graphical user interface (GUI) component container layout manager event event listener event dispatch thread (EDT) thread daemon thread thread safety anonymous class inner class lambda expression functional interface method overriding implementing an interface immutable locale event-driven programming screen coordinates bounding rectangle canvas content pane adapter class RGB color clipping Model-View-Controller (MVC) Observer pattern headless mode

---

9.16 Test Yourself

1. What is the bottom-right pixel painted by the calls `g.drawRect(10, 20, 30, 40)` and `g.fillRect(10, 20, 30, 40)`?

   Answer

   `(40, 60)` for `drawRect` and `(39, 59)` for `fillRect`. The outline also draws the right and bottom edge of the bounding rectangle and so is `w + 1` pixels wide. The filled rectangle paints exactly `w × h` pixels. Both values were found by counting the painted pixels on an image.

2. Which components does `new Color(10, 20, 30).brighter()` give?

   Answer

   `(14, 28, 42)`. Each component is divided by 0.7 and truncated: 10 / 0.7 = 14.28 → 14; 20 / 0.7 = 28.57 → 28; 30 / 0.7 = 42.85 → 42. No component reaches 255, so the cap does not apply.

3. A 300 × 200 pixel panel uses `new BorderLayout(5, 5)`. Components preferring a height of 30 in the north and 20 in the south, a width of 60 in the west and 40 in the east, and one component in the center are added. What are the bounds of the center component?

   Answer

   `(65, 35, 190, 140)`. The center starts to the right of the west region with a 5-pixel gap: `x = 60 + 5`. It starts below the north region with a 5-pixel gap: `y = 30 + 5`. The width is `300 − 60 − 40 − 2 × 5 = 190` and the height is `200 − 30 − 20 − 2 × 5 = 140`. West and east have the same height and are at `y = 35`.

4. The `paintComponent` method of the no-entry sign runs on a 300 × 300 pixel panel. What are `r`, `r2`, `bw`, and `bh`?

   Answer

   `r = min(150, 150) − 5 = 145`. `r2 = (int) 130.5 = 130`. `bw = (int) (130 × 1.4) = 182`. `bh = (int) (130 × 0.35) = (int) 45.5 = 45`. Truncation discards the half as well; with rounding the result would have been 46.

5. What are the counts in the report when 41 is typed into the change program?

   Answer

   0 × 50, 1 × 25, 1 × 10, 1 × 5, 1 × 1. 41 / 50 = 0 (remainder 41), 41 / 25 = 1 (remainder 16), 16 / 10 = 1 (remainder 6), 6 / 5 = 1 (remainder 1), 1 / 1 = 1. Four coins in all.

6. Does the following code compile?

```
String name = " Ayşe ";
name = name.trim();
JButton b = new JButton("x");
b.addActionListener(e -> System.out.println(name));
```

   Answer

   It does not compile: `local variables referenced from a lambda expression must be final or effectively final`. The lambda only reads `name`, but because `name` is reassigned on the second line, it is not effectively final. The fix is to put the trimmed value in a new variable: `String clean = name.trim();` and use `clean` inside the lambda.

7. A `MouseAdapter` object that overrides only `mouseClicked` is registered with a panel using

`addMouseMotionListener`. Does the method run when the panel is clicked?

> **Answer**
>
> It does not. Because `MouseAdapter` implements three interfaces at once, the registration compiles, but the panel recognizes this object only as a motion listener. Click events go only to listeners registered with `addMouseListener`. When a click event was sent to the panel, the method was called 0 times.

8. What does `getLineCount()` return for a `JTextArea` whose text is `"a\n\nb\n"`?

> **Answer**
>
> 4. The text has three line breaks: "a," an empty line, "b," and an empty line after the last line break. The line count is one more than the number of line breaks.

9.17 Further reading

- Oracle, *The Java Tutorials*, "Creating a GUI With Swing": the official, free account of components, layout managers, and event listeners. The lesson "Concurrency in Swing" explains the event dispatch thread in detail.
- Oracle, *The Java Tutorials*, "Performing Custom Painting": shows step by step, with an example, how `paintComponent`, `repaint`, and clipping work together.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals* (12th ed.), Chapters 10–11: covers graphics programming, event handling, and Swing components with an MVC perspective.
- E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*: the original account of the Observer pattern; it explains why Swing listeners are designed the way they are.

9.18 References

1. Oracle, *The Java Tutorials*: Creating a GUI With Swing; Performing Custom Painting; Laying Out Components Within a Container; Writing Event Listeners; Concurrency in Swing.
2. Oracle, *Java SE 21 API Documentation*: `java.awt.Graphics`, `java.awt.Color`, `javax.swing.JFrame`, `javax.swing.SwingUtilities`.
3. P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson (online chapters: Swing GUI Components; Graphics and Java 2D).
4. K. Sierra, B. Bates, and T. Gee, *Head First Java*, 3rd ed., O'Reilly (Chapters 14–15: GUI, Swing, and events).
5. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson (Chapters 10–11: graphics programming, event handling, Swing).
6. A. B. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly (Appendix C: Graphics).

7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

Threads

A program can do several things at once. A word processor checks spelling while you type, a browser stays responsive to one tab while it downloads another, and a server handles hundreds of requests together. Java does this with threads, independent flows of execution inside one program. These notes first introduce threads, processes, concurrency, and parallelism. They then cover the `Thread` class, the thread life cycle, the ways to create and start a thread, and how to stop one safely. The core of the chapter is the protection of shared data: race conditions, `synchronized`, `volatile`, explicit locks, deadlock, and waiting for a condition. The last sections introduce the ready-made tools of the `java.util.concurrent` package, atomic classes, interruption, `ThreadLocal`, and thread pools. The output of a program with several threads often changes from run to run. Every such output in these notes was measured by running the program many times on JDK 21, and each example states which part of the output may vary.

LEARNING OUTCOMES

Concepts
Explain the difference between a thread and a process, and between concurrency and parallelism.

Creating threads
Write a thread by implementing `Runnable` or extending `Thread`, and start it with `start()`.

Life cycle
Trace the six states and the transitions caused by `sleep`, `join`, `wait`, and waiting for a lock.

Synchronization
Recognize a race condition and prevent it with `synchronized`, `volatile`, `Lock`, and atomic classes.

Condition waiting
Write a producer–consumer program with `wait` / `notify` and `Condition`; avoid deadlock.

Ready-made tools
Use `CountDownLatch`, `CyclicBarrier`, `Exchanger`, `Semaphore`, `ThreadLocal`, and thread pools.

10.1 Basic concepts

10.1.1 Threads

Every program written so far ran as a single flow. The first line of `main` runs, then the second; if a method is called, the flow enters that method and resumes where it left off when the method returns. At any moment, the program's "position" can be described by a single line. That single flow is itself a thread.

> **Definition 10.1**
>
> A **thread** is an independently progressing flow of execution inside a program. It executes instructions one after another and has its own call stack and its own program counter. In Java, every thread is represented by an object of the class `java.lang.Thread`.

Every Java program starts with at least one thread. The JVM creates a thread to run `main()`; it is called the **main thread**. If the program starts new threads, they run alongside the main thread. Working with several threads is called **multithreading**. Programming in which several tasks make progress during the same period of time is called **concurrent programming**.

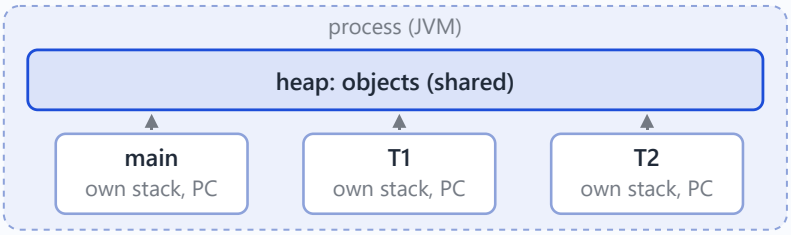


Figure 10.1: Threads in a process. The heap is shared; each thread has its own stack.

The slides compare the situation to a beehive. Each bee does its own job; that is a thread. But all the bees use the same hive and the same honeycomb; that is shared memory. If two bees try to put honey into the same cell at the same moment, there is trouble. Most of this chapter is about how to organize such collisions.

Why do we need several threads? Three reasons stand out:

- **A responsive interface.** In a program with a graphical interface, a long computation done on the thread that draws the interface freezes the window. If the computation is given to a separate thread, the user can keep pressing buttons.

- **Using multicore processors.** Most of today's processors have 4, 8, or more cores. A single-threaded program uses only one of them. If the work is split into pieces, the cores run at the same time and the computation speeds up.
- **Working while waiting.** Waiting for data to arrive over the network or for a file to be read from disk is very slow compared with the processor. While the waiting thread sleeps, another thread carries on with its work.

10.1.2 Concurrency and parallelism

In everyday speech, "at the same time" describes two different situations. In technical language, the two have different names.

Definition 10.2

Concurrency means that several tasks make progress during the same period of time. It can happen even on a single core: the operating system runs threads in turn for very short time slices. **Parallelism** means that tasks really run at the same instant on different cores; it requires more than one core.

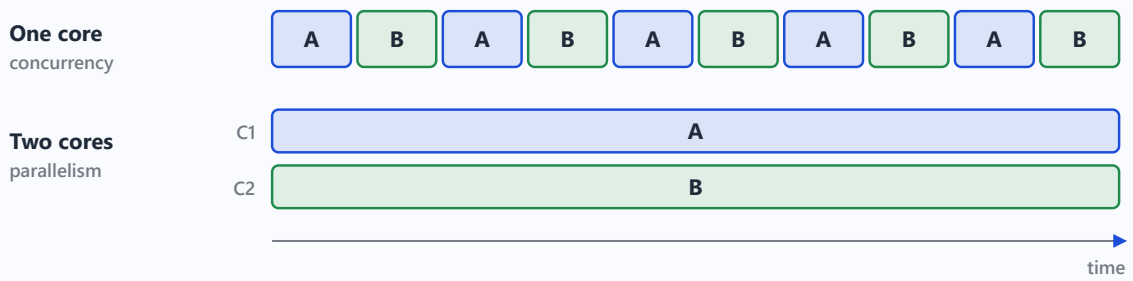


Figure 10.2: Left, one core: A and B run in turn in time slices. Right, two cores: A and B run at the same time.

The left half of Figure 10.2 shows one core. A runs for a slice, then B, then A again. Because the slices last milliseconds, the user believes that both jobs advance at once. This is concurrency but not parallelism: at any given instant, only one thread is on the processor. The right half has two cores, and A and B really run at the same time. Parallelism is a special case of concurrency.

The **scheduler** of the JVM and the operating system decides which thread runs when and on which core. The programmer cannot know or control this order. The same program may run A first in one execution and B first in another. This nondeterminism is the most distinctive feature of multithreaded programs.

10.1.3 Processes and threads

Definition 10.3

A **process** is a running program that has its own memory space. The operating system gives each process a separate address space. Inside a process, one or more threads run; these threads share the process's memory.

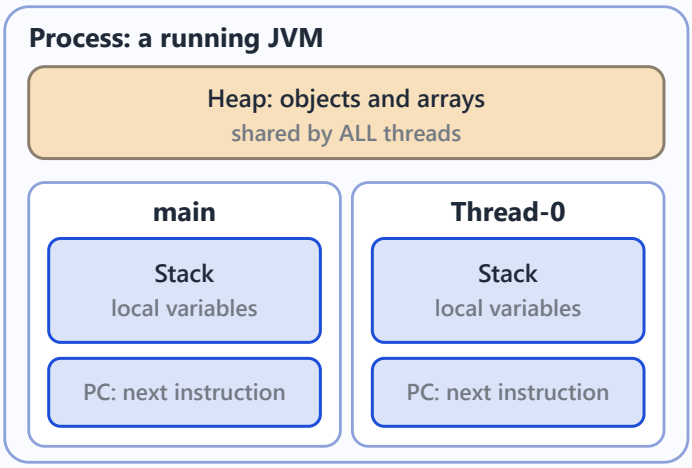


Figure 10.3: One process: a shared heap, and a separate stack and program counter for each thread.

	Process	Thread
Memory	Its own address space	Shares the process's heap
Stack	—	Each has its own
Creation	Expensive	Cheap (lightweight process)
Communication	Files, sockets, pipes	Shared objects
Risk	Cannot corrupt each other	Can corrupt shared data

Table 10.1: Comparison of a process and a thread.

A running Java program is a process, and many threads run inside it. Table 10.1 summarizes the difference between the two concepts. The objects on the heap are open to all threads of the process. If two threads hold references to the same object, both can read and modify it. Local variables, on the other hand, live on each thread's own stack and are not shared. Even if two threads run the same method at the same time, the method's local variable `i` has two separate copies.

This distinction cuts both ways. The good side is that sharing data between threads is very easy: it is enough to write to a shared object. Two processes can send data to each other only through operating-system facilities such as files, sockets, or pipes. The bad side is that two threads that know nothing about each other can modify a shared object at the same time. One process cannot corrupt another's memory; a thread can corrupt anything in its own process.

Creating a thread is much cheaper than creating a process, because no new address space is opened; only a new stack is allocated. That is why a thread is also called a **lightweight process**.

10.1.4 Context switching

A computer usually has many more threads than cores. The operating system runs them in turn in time slices. When a thread's slice ends, its state is saved and the state of the next one is loaded.

Definition 10.4

Context switching is the processor's move from one thread to another. During the switch, the state of the running thread (program counter, registers, stack pointer) is saved and the state of the next thread is loaded.

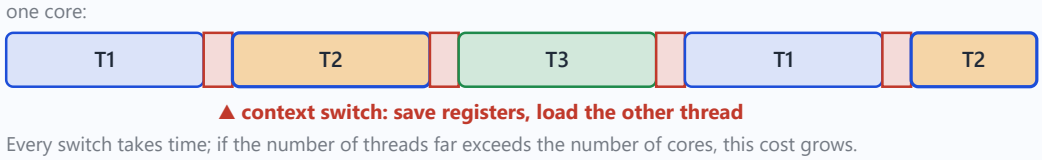


Figure 10.4: On one core, threads run in turn, with a context switch between them.

Context switching has a price. Each switch takes a few microseconds. More importantly, some of the data in the processor's cache is useless to the new thread and has to be read from memory again. For this reason, adding threads does not always speed things up. A program that starts thousands of threads spends a significant part of its time on switches and slows down.

There is a practical rule. For compute-bound work, keep the number of threads at about the number of cores. The core count is available from `Runtime.getRuntime().availableProcessors()`; the examples in these notes were run on a machine with 12 logical cores. For I/O-bound work (network, files), more threads can help because most of them are asleep much of the time. A thread pool keeps the number under control.

10.1.5 The JVM and threads

Java programs do not run directly on the operating system but inside the Java Virtual Machine (JVM). The JVM is a virtual environment that manages threads as independent units of execution. In today's JVMs, every Java thread corresponds to an operating-system thread, and the operating system does the scheduling.

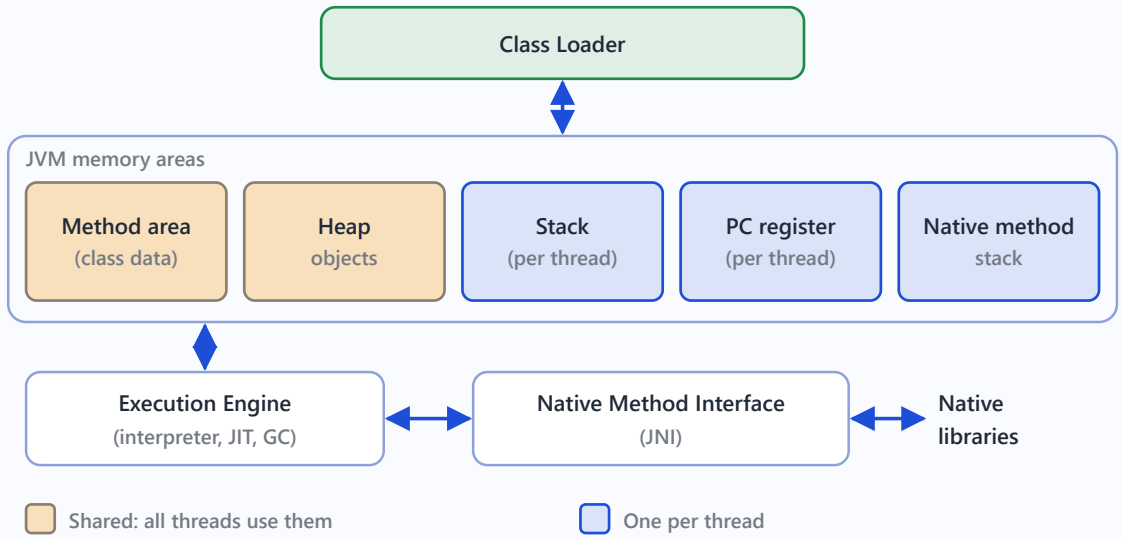


Figure 10.5: JVM architecture. The method area and the heap exist in a single copy; the stack, the PC register, and the native method stack are separate for each thread.

Figure 10.5 shows the memory areas of the JVM. When a thread is created, the JVM gives it three private areas:

- **Stack**: holds the frames of method calls, local variables, and parameters.
- **Program counter (PC) register**: holds the address of the instruction the thread is currently executing.
- **Native method stack**: serves calls to native methods written outside Java, for example in C.

The method area (the code of classes and `static` fields) and the heap (all objects) exist in a single copy and are shared by all threads. A `static` field and an object on the heap are therefore shared between two threads; local variables are not.

Common mistake

Believing that local variables are also shared between threads. If two threads run `void count() { int i = 0; i++; }` at the same time, each increments its own `i` on its own stack; both get 1 and neither affects the other. Trouble begins when two threads write to a field of the same object or to the same `static` field.

10.2 The `Thread` class

10.2.1 The class and its basic methods

Every thread is an object of the class `Thread` in the `java.lang` package. Because the package is `java.lang`, no `import` is needed. The `Thread` object manages the life cycle of the thread: it starts it, makes it wait, and holds its name and state.

One way to write a thread is to write a subclass of `Thread`. The subclass overrides the `run()` method and puts the thread's work there. The thread is then started by calling `start()`. The difference between the two methods is critical: `run()` is the work itself, and `start()` is the call that opens the new flow. Table 10.2 summarizes the frequently used methods of the class.

Method	Purpose
`start()`	Starts the new thread, which then runs `run()`
`run()`	The work the thread does
`sleep(ms)`	`static`: puts the running thread to sleep for a given time
`join()`	Waits for that thread to finish
`yield()`	`static`: temporarily gives up execution to others
`interrupt()`	Sends an interrupt request
`currentThread()`	`static`: the thread that is running right now
`getName()`, `getState()`, `isAlive()`	Read information

Table 10.2: The basic methods of the `Thread` class.

It is significant that `sleep`, `yield`, and `currentThread` are `static`. They act not on a particular `Thread` object but on the thread that executes that line. The thread that writes `Thread.sleep(100)` puts itself to sleep; there is no way to put another thread to sleep.

10.2.2 Thread properties

A `Thread` object carries several properties that describe the thread. They are fields of the object, with get/set methods to read or change them.

Runnable
: The object that contains the `run()` method the thread will execute. The main logic of the thread is defined here.

Name
: Such as `main`, `Thread-0`, `Thread-1`. If no name is given, the JVM appends an increasing number to the prefix `Thread-`. It is used to recognize the thread in log files and while debugging: `getName()`, `setName()`.

ID

A unique number assigned automatically when the thread is created. Since Java 19 it is read with `threadId()`; older versions used `getId()`.

Group (`ThreadGroup`)
: The group the thread belongs to. It is used to organize threads.

Daemon
: A helper thread that serves other threads. The JVM does not wait for it to finish when shutting down.

Context class loader
: The class loader the thread uses when it loads classes at run time.

Priority
: An integer between `MIN_PRIORITY` (1) and `MAX_PRIORITY` (10); the default is `NORM_PRIORITY` (5).

State
: One of `NEW`, `RUNNABLE`, `BLOCKED`, `WAITING`, `TIMED_WAITING`, `TERMINATED`.

Interrupted
: A flag showing whether an interrupt request has arrived; it is set by `interrupt()`.

The following fragment runs inside `main` and prints the properties of the thread that is running at that moment.

```
Thread t = Thread.currentThread();   // current
System.out.println("Name    : " + t.getName());
// threadId(): Java 19+ (getId() before)
System.out.println("ID      : " + t.threadId());
System.out.println("Priority: " + t.getPriority());
String group = t.getThreadGroup().getName();
System.out.println("Group   : " + group);
System.out.println("Daemon  : " + t.isDaemon());
System.out.println("State   : " + t.getState());
```

```
Name    : main
ID      : 1
Priority: 5
Group   : main
Daemon  : false
State   : RUNNABLE
```

`Thread.currentThread()` returns the thread that executes that line. Because the code is in `main`, this is the main thread. Its name is `main`, its ID is 1, and its priority is the default 5. Its group is also named `main`. The main thread is not a daemon thread. Its state is `RUNNABLE`, because the thread calling `getState()` is running at that moment. Run on JDK 21, the program printed these six lines identically every time.

Priority and the daemon property are inherited from the thread that creates the new thread. A thread created in `main` has priority 5 and is not a daemon. These values can be changed before `start()`.

Example 10.1

In `main`, before any other thread has been created, these lines run: `Thread a = new Thread(() -> { }, "worker");`, `Thread b = new Thread(() -> { });`, `Thread c = new Thread(() -> { });`, and `c.setPriority(Thread.MAX_PRIORITY);`. What are the names of `a`, `b`, and `c`, and the priorities of `b` and `c`?

The names are `worker`, `Thread-0`, and `Thread-1`. A thread that is given a name does not use up an automatic number; numbers go in order only to unnamed threads. The priority of `b` is 5, inherited from the creating thread (`main`); that of `c` is 10. None of the three was ever started, so their states are `NEW`. On JDK 21 the program printed `worker Thread-0 Thread-1` and `5 10 false`.

10.2.3 Thread groups

When the JVM starts, it creates two thread groups. The `system` **group** contains the JVM's own threads: `Reference Handler`, which manages references, the finalizer `Finalizer`, `Signal Dispatcher`, which handles signals, and others. The `main` **group** is a subgroup of `system`; the main thread and the threads the program creates belong to it by default.

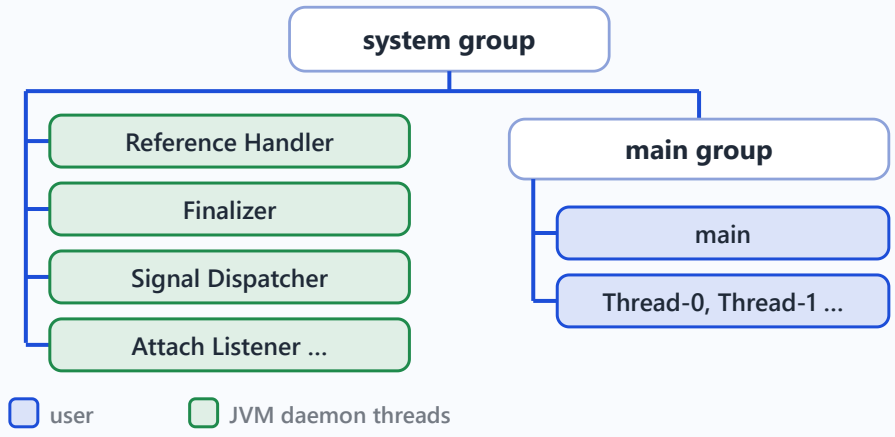


Figure 10.6: Thread groups: the JVM threads and the `main` group under the `system` group.

`Thread.getAllStackTraces()` returns all live threads and their stacks as a map. Iterating over the key set lists them all:

```
Set<Thread> all =
        Thread.getAllStackTraces().keySet();
for (Thread t : all) {
    System.out.println(t.getThreadGroup().getName()
            + "  " + t.getName()
            + "  daemon=" + t.isDaemon());
}
```

```
system  Notification Thread  daemon=true
main  main  daemon=false
system  Signal Dispatcher  daemon=true
system  Reference Handler  daemon=true
system  Attach Listener  daemon=true
InnocuousThreadGroup  Common-Cleaner  daemon=true
system  Finalizer  daemon=true
```

This output was taken from one run on JDK 21. The list has seven threads, and all except `main` are daemon threads. `Common-Cleaner` is in a separate group, `InnocuousThreadGroup`. The order of the lines depends on the internal layout of the map. In ten runs the order was always the same, but that is not a guarantee. Which JVM threads appear in the list also varies with the JVM version and settings.

10.2.4 Threads running when the JVM starts

The slides discuss four of the threads that exist when the HotSpot JVM starts:

Main thread
: The first thread that runs when the JVM starts. The `main()` method runs here; the core logic of the application executes here.

Garbage collector
: Responsible for memory management. It finds and cleans up objects that no longer have references. In the HotSpot JVM, the garbage collector's threads are internal JVM threads; they do not appear as Java objects, so they are not in the list above.

Signal Dispatcher
: Handles signals from the operating system. For example, on a shutdown request that arrives with Ctrl+C, it starts the orderly shutdown of the JVM.

Finalizer
: Calls the `finalize()` method of objects. `finalize()` was deprecated in Java 9 and marked for removal in Java 18 (JEP 421). If the JVM is started with the `--finalization=disabled` option, the finalizer thread never starts. Use `try-finally` or try-with-resources with `AutoCloseable` instead.

10.3 The life cycle

10.3.1 The six states

From the moment it is created until it finishes, a thread passes through different states. The `getState()` method returns the current state as a constant of the `Thread.State` enumeration. Java defines six states.

State	Meaning	How it is entered
`NEW`	The object has been created but not yet started	`new Thread(…)`; exit: `start()`
`RUNNABLE`	Running, or waiting for its turn on the processor	After `start()`
`BLOCKED`	Waiting for a monitor lock to enter a `synchronized` block or method	While another thread holds the lock
`WAITING`	Waiting indefinitely; another thread must wake it	`Object.wait()`, `Thread.join()`, `LockSupport.park()`
`TIMED_WAITING`	Waiting for a limited time	`Thread.sleep(ms)`, `wait(ms)`, `join(ms)`, `parkNanos`, `parkUntil`
`TERMINATED`	Has completed execution	When `run()` returns

Table 10.3: Thread states.

Table 10.3 makes one point clear: Java has no separate RUNNING state. The thread that is on the processor right now and a thread waiting for its turn are both `RUNNABLE`. The scheduler knows which one is really on the processor; the JVM does not expose this distinction to the program.

The three waiting states should not be confused. `BLOCKED` means only waiting for a monitor lock: the thread wants to enter a `synchronized` block, but the lock is held by another thread. `WAITING` is waiting indefinitely for an event: another thread must finish (`join`) or wake it (`notify`). `TIMED_WAITING` is waiting for a limited time; when the time runs out, the thread returns on its own. That is why `sleep` always produces `TIMED_WAITING`.

10.3.2 The state diagram

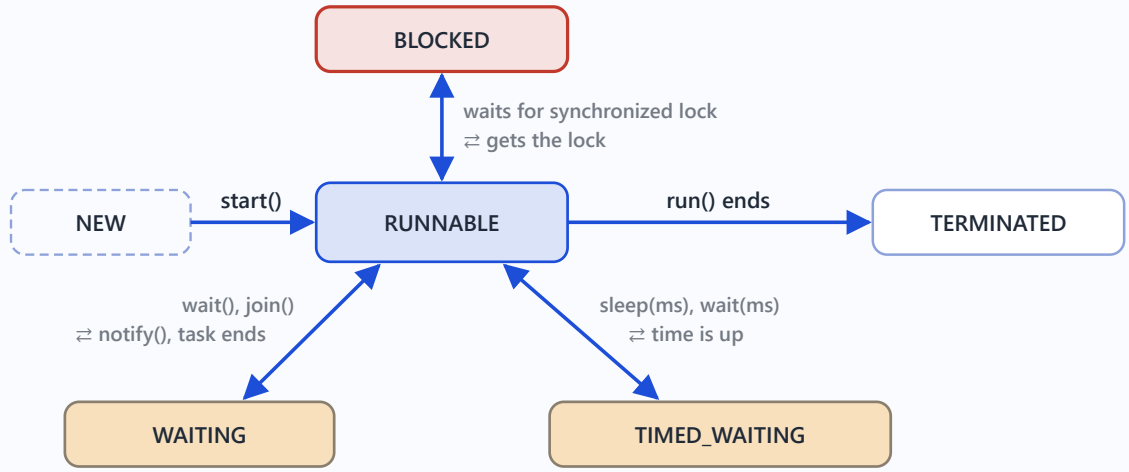


Figure 10.7: The state diagram. `NEW` becomes `RUNNABLE` with `start()`; there are round trips between `RUNNABLE` and the three waiting states; when `run()` ends, the thread is `TERMINATED`.

Figure 10.7 shows the transitions. The main path is one-way: `NEW` → `RUNNABLE` → `TERMINATED`. There is no way back. A thread that has finished cannot be restarted; to do the same work again, a new `Thread` object is needed. From each of the three waiting states, on the other hand, the thread returns to `RUNNABLE`: when the lock is free, when the awaited event happens, or when the time runs out.

10.3.3 Code trace: observing the states

The following program puts four threads into different states and prints their states. `pause(ms)` and `waitFor(t)` are helper methods that wrap the calls `Thread.sleep` and `t.join()` in `try`-`catch`. `lock` is a field of type `static final Object`.

```
public static void main(String[] args) {
    Thread sleeper = new Thread(() -> pause(1000));
    System.out.println("1) " + sleeper.getState());
    sleeper.start();
    pause(100);
    System.out.println("2) " + sleeper.getState());
    Thread waiter = new Thread(() -> waitFor(sleeper));
    waiter.start();
    pause(100);
    System.out.println("3) " + waiter.getState());
    Thread blocked = new Thread(() -> {
        synchronized (lock) { }
    });
    synchronized (lock) {
        blocked.start();
        pause(100);
        System.out.println("4) " + blocked.getState());
    }
    waitFor(waiter);
    System.out.println("5) " + sleeper.getState());
}
```

Step	Line	Variables / state	New output
1	2: `Thread sleeper = new Thread(() -> pause(1000));`	—	—
2	3: `System.out.println("1) " + sleeper.getState());`	—	`1) NEW`
3	4: `sleeper.start();`	—	—
4	5: `pause(100);`	—	—
5	6: `System.out.println("2) " + sleeper.getState());`	—	`2) TIMED_WAITING`
6	8: `waiter.start();`	—	—
7	10: `System.out.println("3) " + waiter.getState());`	—	`3) WAITING`
8	15: `blocked.start();`	—	—
9	17: `System.out.println("4) " + blocked.getState());`	—	`4) BLOCKED`
10	19: `waitFor(waiter);`	—	—
11	20: `System.out.println("5) " + sleeper.getState());`	—	`5) TERMINATED`

Table 10.4: Trace of the state-observation program.

In steps 1 and 2, the `sleeper` object is created but not started; its state is `NEW`. In step 3, `start()` is called. The new thread immediately enters `pause(1000)` and sleeps for a second. Meanwhile `main` also sleeps for 100 ms. When `main` wakes up, `sleeper` is still asleep: `TIMED_WAITING`.

In step 6, `waiter` is started and calls `sleeper.join()`. `join` without a timeout is indefinite; `waiter` stays in the `WAITING` state until `sleeper` finishes. In step 8, `main` acquires the monitor of `lock` and then starts `blocked`. `blocked` wants to enter a `synchronized` block on the same lock, but `main` holds it. While `main` sleeps for 100 ms inside the block, it prints the state: `BLOCKED`.

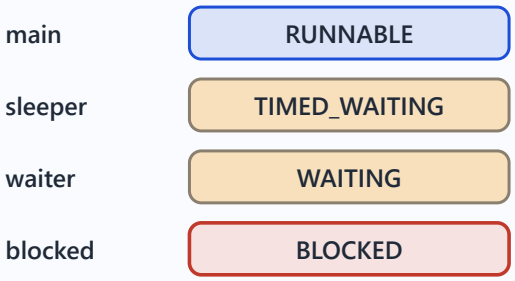


Figure 10.8: Step 9. `main` holds the lock and sleeps; `sleeper` is asleep, `waiter` is waiting for it, and `blocked` is waiting for the lock.

Figure 10.8 shows the situation in step 9: four threads are in four different states. One point deserves attention. While `main` sleeps inside `sleep`, it does not release the lock, so `blocked` cannot get in. When the block ends, the lock is released, `blocked` enters, leaves immediately, and finishes. `main` then waits for `waiter`. `sleeper` finishes its full second and ends; `waiter` then finishes too. The last line prints the state of `sleeper`: `TERMINATED`.

Although this program is multithreaded, its output is stable. The waiting times determine which state the threads will be in at each observation: 100 ms is enough for a thread to start and enter `sleep` or `join`. The program was run 8 times and printed the same five lines each time. Even so, such timing assumptions can break on a heavily loaded machine. The delays are used here only for observation, not for synchronization.

10.4 Creating and starting threads

10.4.1 The `Runnable` interface

Every thread runs a `Runnable` object. The `Runnable` interface is in the `java.lang` package and contains a single method:

```
package java.lang;

public interface Runnable {
    void run();
}
```

A class that implements this interface writes the `run()` method. The body of the method is the work the thread will do.

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        // The thread's tasks
    }
}
```

Because `Runnable` has a single abstract method, it is a functional interface. Since Java 8, a lambda expression can be written in place of such an interface. A separate class is not needed for short tasks:

```
Runnable r = () -> System.out.println("hello");
```

The lambda gives the body of the `run()` method directly. Its parameter list is empty because `run()` takes no parameters, and since the return type is `void`, the lambda returns no value.

10.4.2 Starting a thread

A `Runnable` object is not a thread by itself; it only describes the work to be done. To run the work in a new thread, the object is passed to the `Thread` constructor and `start()` is called.

```
public class Worker implements Runnable {
    // The thread's work goes here
    public void run() {
        String name = Thread.currentThread().getName();
        System.out.println("Worker running: " + name);
    }
    public static void main(String[] args) {
        // Start the thread
        Worker w = new Worker();
        Thread t = new Thread(w);
        t.start();
        System.out.println("main goes on");
    }
}
```

The slides show the output of one run like this:

```
main goes on
Worker running: Thread-0
```

`main` creates the `Worker` object and passes it to the constructor to create a `Thread`. `t.start()` opens the new thread and returns immediately. From then on there are two threads: `main` moves on to the next line and prints `main goes on`, while `Thread-0` runs `run()` and prints its own name. The

name was given automatically: the first unnamed thread the program creates is `Thread-0`.

Which of the two lines is printed first is not determined. As soon as `start()` returns, `main` continues; the scheduler decides when the new thread gets the processor. The program was run 40 times on JDK 21, and `main goes on` came first every time. The reverse order is also valid and can be seen on another machine or under load. The measured distribution varies with the machine and the load; the correctness of a program must never depend on any particular order.

> **Common mistake**
>
> Calling `run()` instead of `start()`. `t.run()` is an ordinary method call: the code runs in the caller's thread, that is, in `main`, and no new thread is opened. The program raises no error, but nothing runs in parallel. For this reason the mistake often goes unnoticed.

10.4.3 Code trace: two stacks, two flows

Tracing the same program step by step shows what `start()` really does. The trace shows one of the possible orders, the one in which `main` prints first.

Step	Line	Variables / state	New output
1	9: `Worker w = new Worker();`	w=Worker object, t=…	—
2	10: `Thread t = new Thread(w);`	w=Worker object, t=Thread-0 (NEW)	—
3	11: `t.start();`	w=Worker object, t=Thread-0 (RUNNABLE)	—
4	12: `System.out.println("main goes on");`	w=Worker object, t=Thread-0 (RUNNABLE)	`main goes on`
5	13: `}`	t=Thread-0 (RUNNABLE)	—
6	4: `String name = Thread.currentThread().getName();`	t=Thread-0 (RUNNABLE), name="Thread-0"	—
7	5: `System.out.println("Worker running: " + name);`	t=Thread-0 (RUNNABLE), name="Thread-0"	`Worker running: Thread-0`
8	6: `}`	t=Thread-0 (TERMINATED)	—

Table 10.5: One possible order of the `Worker` program.

In step 2 the `Thread` object has been created and its state is `NEW`. But there is not yet a separate flow of execution; the object is an ordinary object waiting on the heap. In step 3, `start()` is called. The JVM opens a new stack and starts `run()` on that stack. `start()` itself returns without waiting for `run()` to finish.



Figure 10.9: Step 6. `main()` has finished, but `Thread-0` is running `run()` on its own stack. `name` is its local variable.

In step 5, the `main()` method ends. The program does not. The JVM keeps running until all threads that are not daemon threads have finished. Figure 10.9 shows this moment: the stack of `main` is empty, and the stack of `Thread-0` holds the frame of `run()`. The variable `name` is in this frame, and no other thread can see it. In step 8, `run()` ends and `Thread-0` becomes `TERMINATED`. With no running threads left, the JVM exits.

10.4.4 Extending `Thread`

A thread can also be written by extending the `Thread` class. `Thread` already implements `Runnable`; the subclass only overrides `run()`.

```
public class Worker2 extends Thread {
    // The thread's work goes here
    public void run() {
        String name = getName();   // inherited from Thread
        System.out.println("Worker running: " + name);
    }
    public static void main(String[] args) {
        // Create and start the thread
        Worker2 w = new Worker2();
        w.start();
        System.out.println("main goes on");
    }
}
```

```
main goes on
Worker running: Thread-0
```

In this approach the `Runnable` object and the `Thread` object are the same. Because the object is itself a `Thread`, `w.start()` is called directly. Inside `run()`, `getName()` is written without a prefix because the method is inherited from `Thread`. The output order may again vary: in 40 runs, `main` printed first every time.

10.4.5 `start()` or `run()`?

The difference between `start()` and `run()` becomes visible when both are called one after the other on the same object.

```
public class StartVsRun {
    public static void main(String[] args)
            throws InterruptedException {
        Thread t = new Thread(() -> {
            String name = Thread.currentThread().getName();
            System.out.println("inside run(): " + name);
        });
        t.run();     // WRONG: main runs it
        t.start();   // RIGHT: a new thread
        t.join();
    }
}
```

```
inside run(): main
inside run(): Thread-0
```

The first line comes from `t.run()`. This call is an ordinary method call inside `main`; `currentThread()` returns `main`. The second line comes from `t.start()`: now `run()` runs in the new thread, `Thread-0`. The order of the output is fixed: execution does not reach the `t.start()` line until the `t.run()` call has finished. `join()` makes `main` wait until `Thread-0` ends.

Call	What happens
`t.run()`	An ordinary method call: runs in the same thread (`main`), no new stack, and `main` waits until it finishes.
`t.start()`	The JVM creates a new thread and stack; `run()` runs there, and `start()` returns immediately.
`t.start()` a second time	`IllegalThreadStateException` : a `Thread` object can be started only once.

Table 10.6: The difference between calling `run()` and `start()`.

The rule is short: the thread's work goes into `run()`, and the thread is started with `start()`. If `start()` is called a second time on the same object, JDK 21 threw `java.lang.IllegalThreadStateException` both while the thread was running and after it had finished. The reason is the one-way path in the state diagram: `start()` is valid only for a thread in the `NEW` state.

Example 10.2

A thread named `T` is created; its job is to print `"run: " + Thread.currentThread().getName()`. Then these lines run in order: `t.run();`, `System.out.println("A " + t.getState());`, `t.start();`, `t.join();`, and `System.out.println("B " + t.getState());`. What does the program print?

The call `t.run()` runs in `main`: `run: main`. The thread has still never been started: `A NEW`. After `start()`, `run()` runs on `T`'s own stack: `run: T`. `join()` waits until `T` ends, so the last line is `B TERMINATED`. Because `join()` fixes the order, these four lines are the same in every run; the program was run five times and always gave the same output.

10.4.6 The single-inheritance problem

Java allows a class to inherit from only one class. If the class that should run in a thread already extends another class, it cannot extend `Thread`. Writing `class Worker extends Base, Thread` is a compile-time error; `javac` reports `'{' expected` at the comma.

The solution lies in the `Runnable` interface. There is no limit on the number of interfaces a class can implement. The class extends `Base`, implements `Runnable`, and, if needed, keeps a field of type `Thread` inside.

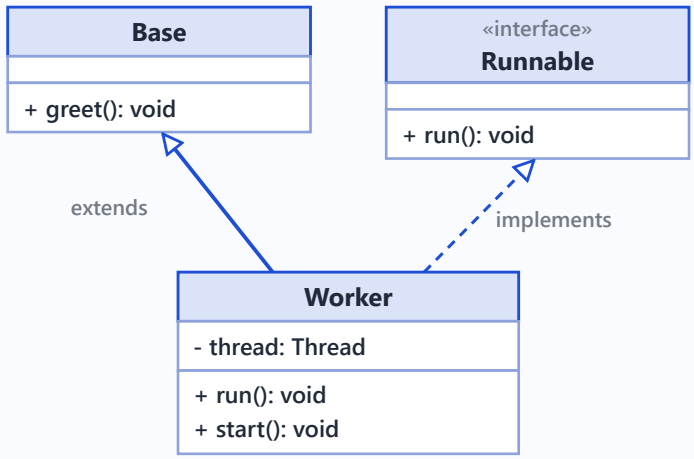


Figure 10.10: `Worker` extends `Base` and implements `Runnable`; it has a `Thread` field named `thread`.

```
class Base {
    public void greet() {
        System.out.println("Hello from Base");
    }
}
class Worker extends Base implements Runnable {
    private Thread thread;   // Thread as a field
    public void run() {
        String name = Thread.currentThread().getName();
        System.out.println("run(): " + name);
    }
    public void start() {
        thread = new Thread(this);
        thread.start();
    }
}
public class SingleInheritance {
    public static void main(String[] args) {
        Worker w = new Worker();
        w.greet();   // inherited from Base
        w.start();   // our own start()
    }
}
```

```
Hello from Base
run(): Thread-0
```

`Worker` both inherits the methods of `Base` and can run in a thread. `w.greet()` is the method that comes from `Base`, and it runs in `main`. `w.start()` is not the `start()` of `Thread` but a `start()` method that `Worker` defines itself. It hands itself to a `Thread` with `new Thread(this)` and starts it. `this` is of type `Runnable`, which is why the `Thread` constructor accepts it. The order of the two output lines is fixed: `greet()` finishes before the thread is started.

This pattern separates the work from the thread. `Worker` describes "what to do", and the `Thread` object describes "how to run it". The same `Runnable` object can later be handed to a thread pool as well. This is why `Runnable` is preferred in most cases.

10.4.7 Comparing the three ways

The following program puts the three ways of creating a thread side by side. A `static class` is a class defined inside another class; it can be used without an instance of the outer class.

```
public class ThreeWays {
    static class Task implements Runnable {     // 1)
        public void run() {
            System.out.println("1) Runnable class");
        }
    }
    static class MyThread extends Thread {       // 2)
        public void run() {
            System.out.println("2) Thread subclass");
        }
    }
    public static void main(String[] args)
            throws InterruptedException {
        Thread t1 = new Thread(new Task());
        Thread t2 = new MyThread();
        // 3) Lambda (Java 8+)
        Thread t3 = new Thread(
                () -> System.out.println("3) Lambda"));
        t1.start(); t1.join();
        t2.start(); t2.join();
        t3.start(); t3.join();
    }
}
```

```
1) Runnable class
2) Thread subclass
3) Lambda
```

Because each `start()` is followed immediately by `join()`, one thread does not begin before the previous one has finished. The output order is therefore fixed. Of course, this usage defeats the whole point of threads; the program was written only to show the three ways.

Way	When to use it
`implements Runnable`	When the class already extends another class; keeps the work and the thread separate
`extends Thread`	Short, self-contained examples; uses up the one inheritance slot
Lambda (Java 8+)	The most practical way for short tasks

Table 10.7: Three ways to create a thread.

10.4.8 Code trace: two threads

How does the output come about when two threads really run together? In the following program, two threads named `A` and `B` run the same `task()` method. The second argument of the `Thread` constructor is the thread's name.

```
public class TwoThreads {
    static void task() {
        String name = Thread.currentThread().getName();
        for (int i = 1; i <= 2; i++) {
            System.out.println(name + ": step " + i);
        }
    }
    public static void main(String[] args)
            throws InterruptedException {
        Thread a = new Thread(() -> task(), "A");
        Thread b = new Thread(() -> task(), "B");
        a.start();
        b.start();
        a.join();
        b.join();
        System.out.println("main: done");
    }
}
```

Step	Line	Call stack	Variables / state	New output
1	12: `a.start();`	main()	a=RUNNABLE, b=NEW	—
2	13: `b.start();`	main()	a=RUNNABLE, b=RUNNABLE	—
3	14: `a.join();`	main()	a=RUNNABLE, b=RUNNABLE, main=WAITING	—
4	5: `System.out.println(name + ": step " + i);`	A: run() → task()	A.i=1, B.i=—, main=WAITING	`A: step 1`
5	5: `System.out.println(name + ": step " + i);`	B: run() → task()	A.i=1, B.i=1, main=WAITING	`B: step 1`
6	5: `System.out.println(name + ": step " + i);`	A: run() → task()	A.i=2, B.i=1, main=WAITING	`A: step 2`
7	5: `System.out.println(name + ": step " + i);`	B: run() → task()	A.i=2, B.i=2, main=WAITING	`B: step 2`
8	16: `System.out.println("main: done");`	main()	A=TERMINATED, B=TERMINATED, main=RUNNABLE	`main: done`

Table 10.8: One possible order of the two threads (the order in the slides).

In steps 1 and 2, `main` starts the two threads. Both are `RUNNABLE`, but which one will run when is not determined. In step 3, `main` begins waiting for `A` to finish with `a.join()` and becomes `WAITING`. Now it is `A` and `B` that are running.

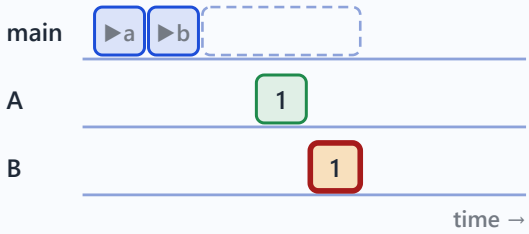


Figure 10.11: Step 5. A context switch: `B` is running now. The `i` of `B` is a separate local variable from the `i` of `A`.

In steps 4–7, `A` and `B` each print one line in turn. At every context switch, the other thread resumes where it left off. The `i` of `A` and the `i` of `B` are on separate stacks; incrementing one does not affect the other. In step 8, `a.join()` returns. Because `B` has already finished, `b.join()` returns without waiting, and `main` prints the last line.

This order is only one of the possible ones. The program was run 60 times on JDK 21; the outputs seen and their counts:

```
32 times:  B: step 1 · B: step 2 · A: step 1 · A: step 2 · main: done
25 times:  A: step 1 · A: step 2 · B: step 1 · B: step 2 · main: done
 3 times:  B: step 1 · A: step 1 · A: step 2 · B: step 2 · main: done
```

When the same two threads were started repeatedly inside a single JVM, the picture changed: in 6,000 trials, A1 · A2 · B1 · B2 occurred 5,958 times, B1 · B2 · A1 · A2 35 times, A1 · B1 · A2 · B2 5 times, and A1 · B1 · B2 · A2 2 times. In a warmed-up JVM, a new thread starts very quickly and finishes its short job before the other one even starts. The order in the slides was indeed seen, though rarely. Two things are certain. First, `main: done` is always last, because it is printed after both `join()` calls. Second, the two lines of the same thread keep their order relative to each other: `A: step 2` never comes before `A: step 1`. The lines of the two threads, however, can interleave in any way.

Example 10.3

In `main` these run in order: start `a`, `a.join()`, start `b`, print `"m"`, `b.join()`. Thread `a` prints `"a"` and `b` prints `"b"`. Which outputs are possible?

`a.join()` does not let `main` proceed until `a` has finished, so `a` is always the first letter. After `b` is started, `main` printing `m` races with `b` running. The possible outputs are `abm` and `amb`; `mab` is impossible. `b.join()` only prevents the program from ending before `b`. In five rounds of 300 trials each inside a single JVM, 1,500 trials in all, every one gave `amb`; in 40 runs with separate JVMs, `amb` was seen every time as well. By the time the new thread starts, `main` has usually already printed its line. Still, `abm` is a valid order; no order is guaranteed.

10.5 Managing and stopping threads

10.5.1 `sleep`, `join`, and `yield`

Three basic methods control the flow of a thread.

`sleep(ms)`
: Puts the running thread to sleep for a given time; during this time the thread is in the `TIMED_WAITING` state. A sleeping thread does not release the locks it holds.

`join()`
: The calling thread waits until the target thread has finished. The form `join(ms)` waits at most `ms` milliseconds; when the time is up it returns even if the target has not finished.

`yield()`
: A hint that the thread is willing to give up execution temporarily to another thread. The scheduler may ignore the hint, so no order is guaranteed with `yield()`.

The following program shows why `join()` is needed. The `calc` thread adds the numbers from 1 to 1,000,000 and writes the result into a `static` field.

```
public class JoinDemo {
    static long result = 0;
    public static void main(String[] args)
            throws InterruptedException {
        Thread calc = new Thread(() -> {
            long sum = 0;
            for (int i = 1; i <= 1_000_000; i++) {
                sum += i;
            }
            result = sum;
        });
        calc.start();
        System.out.println("before join: " + result);
        calc.join();   // wait for calc to end
        System.out.println("after join : " + result);
    }
}
```

```
before join: 0
after join : 500000500000
```

Right after `start()`, `main` reads `result`. The computation has not yet finished at that point; `main` sees the initial value, 0. The program was run 30 times, and the first line was 0 every time. This is not a guarantee: a `main` that was slow to continue, or a very fast computation, could have shown the

result on the first line as well. After `join()`, however, the situation is certain. The computation has finished, and the value it wrote is visible to `main`: $1 + 2 + \ldots + 1{,}000{,}000 = 1{,}000{,}000 \cdot 1{,}000{,}001 / 2 = 500{,}000{,}500{,}000$.

It matters that the result is kept in a `long`. The total is about $5 \cdot 10^{11}$, far above 2,147,483,647, the upper limit of `int`.

`join()` gives a second guarantee: everything the target thread wrote is visible to the thread that returns from `join()`. That is why it is not a problem that `result` is not `volatile`.

`sleep` and `join` can throw `InterruptedException`, a checked exception. Wherever they are called, they must therefore be either caught or declared with `throws InterruptedException`. Most examples in the slides declare `main` with `throws InterruptedException`. Inside a lambda, `throws` cannot be written, because `run()` declares no checked exceptions; there `try-catch` is mandatory.

Common mistake

Writing `Thread.sleep(100);` without `try-catch` in a method that does not declare `throws`. The compiler reports `unreported exception InterruptedException; must be caught or declared to be thrown`. The fix is a `try-catch` or adding `throws InterruptedException` to the method signature.

10.5.2 Daemon threads

Definition 10.5

A **daemon thread** is a helper thread that does not prevent the JVM from shutting down. The JVM exits when all threads that are not daemon threads have finished; it does not wait for daemon threads to finish.

```
public class DaemonDemo {
    public static void main(String[] args)
            throws InterruptedException {
        Thread clock = new Thread(() -> {
            int s = 0;
            while (true) {
                System.out.println("tick " + s++);
                try {
                    Thread.sleep(300);
                } catch (InterruptedException e) {
                    return;
                }
            }
        });
        clock.setDaemon(true);   // BEFORE start()
        clock.start();
        Thread.sleep(1000);
        System.out.println("main done, JVM exiting");
    }
}
```

```
tick 0
tick 1
tick 2
tick 3
main done, JVM exiting
```

`clock` is in an infinite loop and prints a line every 300 ms. `setDaemon(true)` makes it a daemon thread. `main` sleeps for one second, prints its last line, and ends. Only a daemon thread is left, so the JVM exits; the loop of `clock` is cut off midway.

How many `tick` lines are printed? `clock` prints at about 0, 300, 600, and 900 ms; its fourth sleep would end at 1200 ms. `main` wakes up at 1000 ms. The expected output is therefore four `tick` lines. The program was run 10 times and printed four `tick` lines every time. Like any output that depends on timing, this is not guaranteed; the margin between 900 ms and 1000 ms is small.

`setDaemon(true)` must be called before `start()`. If it is called after the thread has started, `IllegalThreadStateException` is thrown; this behavior was tried on JDK 21. Daemon threads are meant for helper work such as logging, cache cleanup, or periodic checks. A job that writes to a file or must not be interrupted is not given to a daemon thread, because when the JVM exits, even `finally` blocks may be cut off without running.

10.5.3 Stopping a thread

A thread stops when its `run()` method completes. The JVM exits in two cases: when `System.exit()` (internally `Runtime.exit()`) is called, or when all threads that are not daemon threads have finished.

The right way to stop a thread from outside is cooperation. The thread to be stopped checks a flag at regular intervals; when the flag is raised, it finishes its work cleanly and leaves `run()`. The flag is declared `volatile` so that a change made by another thread becomes visible. A second way is to send an interrupt request with `interrupt()`.

Older versions of Java had a method called `Thread.stop()`. It killed the thread in the middle of its work. The locks the thread held were suddenly released, and the objects they protected were left half updated. For example, money might have been withdrawn from one account but not yet deposited into the other. The method stayed deprecated for a long time; since Java 20 it no longer works:

```
Thread t = new Thread(() -> { });
try {
    t.stop();
} catch (UnsupportedOperationException e) {
    System.out.println("stop(): not supported");
}
```

```
stop(): not supported
```

On JDK 21, `stop()` always throws `UnsupportedOperationException`. In the same experiment, `suspend()` and `resume()` threw the same exception.

10.5.4 Stopping with a flag

```
class Worker extends Thread {
    private volatile boolean flag = false;
    public void finish() { flag = true; }
    public void run() {
        while (!flag) {
            System.out.println("working...");
            try { Thread.sleep(100); }
            catch (InterruptedException e) { return; }
        }
        System.out.println("flag seen, finishing");
    }
}
public class FlagStop {
    public static void main(String[] args)
            throws InterruptedException {
        Worker w = new Worker();
        w.start();
        Thread.sleep(350);
        w.finish();   // raise the flag
        w.join();     // wait for it to end
        System.out.println("main: worker stopped");
    }
}
```

```
working...
working...
working...
working...
flag seen, finishing
main: worker stopped
```

`finish()` only raises the flag; it does not stop the thread by force. The worker checks the condition `!flag` at the start of every iteration. On the first iteration in which it sees the flag raised, it leaves the loop, prints its last line, and returns from `run()` by itself. `main` waits for the worker to finish with `join()`.

The timing is as follows: the worker prints at about 0, 100, 200, and 300 ms. `main` wakes up at 350 ms and raises the flag. When the worker wakes at 400 ms, it sees the flag. The program was run 15 times, and four `working...` lines were seen every time. The count depends on timing; the correctness of the flag method does not depend on the count.

What would happen if the flag were not `volatile`? The worker could keep the value it first read in a cache or a register and never see the change. In this example, the `sleep` call in the loop often hides the problem; in an empty loop, the thread could really get stuck.

10.5.5 Stopping through a reference

A second form of cooperative stopping is to keep a reference to the thread itself instead of a flag.

```
class Worker implements Runnable {
    private volatile Thread thread;
    public void start() {
        thread = new Thread(this);
        thread.start();
    }
    public void finish() { thread = null; }
    public void run() {
        Thread me = Thread.currentThread();
        while (thread == me) {   // ... work ...
        }
        System.out.println("run() done");
    }
}
public class ReferenceStop {
    public static void main(String[] args) throws Exception {
        Worker w = new Worker();
        w.start();
        Thread.sleep(200);
        w.finish();
        Thread.sleep(200);
        System.out.println("main: done");
    }
}
```

```
run() done
main: done
```

The class implements `Runnable` and keeps the running `Thread` object in a `volatile` field. The loop continues as long as the field still points to its own thread. The comparison with `Thread.currentThread()` checks this. `finish()` sets the field to `null`; the condition fails and `run()` ends. `main` stops the worker after 200 ms, waits another 200 ms, and prints its last line. In ten runs, the output was always these two lines.

One detail: `currentThread()` is a `static` method of `Thread`. In a class that does not extend `Thread`, it must be written with the `Thread.` prefix. If the prefix is forgotten, the compiler reports `cannot find symbol`. Because the body of the loop is empty, the thread keeps one core completely busy for these 200 ms. In a real program, the body does some work or sleeps for a short time.

10.6 Synchronization

10.6.1 Why the order breaks

When several threads work with the same data, performing operations in the right order is hard for three reasons.

Speed is unpredictable
: The speed and frequency at which each thread runs vary. Operating systems may ignore priorities, so priority cannot be trusted for ordering.

Values can go stale
: After one thread reads a variable, another can update the value. The first keeps working with an out-of-date value.

Instructions can be reordered
: The compiler, the JIT (just-in-time) compiler, and the processor may change the order of instructions for speed. In a single thread this goes unnoticed; with more than one, it causes unexpected behavior.

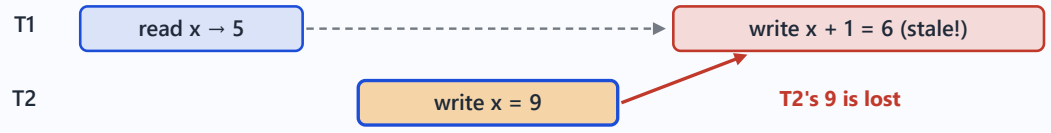


Figure 10.12: A stale value: T1 reads, then T2 writes; T1 continues with the old value.

If several threads access shared, changing data, the access must be synchronized. Otherwise the result depends on the order in which the threads run.

Definition 10.6

A **race condition** occurs when the result of a program depends on the order in which threads run. **Synchronization** is the coordination of the threads' access to shared data so that the result no longer depends on that order.

10.6.2 Race conditions

The simplest race example is two threads incrementing the same counter. Each thread does `counter++` 100,000 times; the expected result is 200,000.

```
public class RaceCondition {
    static int counter = 0;   // shared data
    public static void main(String[] args)
            throws InterruptedException {
        Runnable inc = () -> {
            for (int i = 0; i < 100_000; i++) {
                counter++;   // read, add 1, write: NOT atomic
            }
        };
        Thread t1 = new Thread(inc);
        Thread t2 = new Thread(inc);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Expected: 200000");
        System.out.println("Actual  : " + counter);
    }
}
```

Three runs in the slides gave 138001, 135352, and 136776. For these notes the program was run 20 more times on JDK 21. The values found:

```
136210, 145954, 155340, 149684, 132748, 125554, 126412, 156356, 126897, 144040, 137525,
127748, 117573, 164660, 129167, 134217, 140315, 129857, 128366, 133715
(smallest 117573, largest 164660)
```

Every run gave a different result, and none reached 200,000. Some of the increments are lost. The amount lost varies completely from run to run; it depends on the number of cores in the machine, the load at that moment, and when the JIT compiles the code. In a program that makes few increments (say 1,000), the result is often correct, because the threads may finish without ever overlapping. The bug therefore looks random and is hard to catch by testing.

10.6.3 Code trace: why does `counter++` lose updates?

In the source code, `counter++` looks like a single operation. On the processor it is three steps: read the value from memory, add one, write it back to memory.

```
// counter++ is three CPU steps:
int reg = counter;     // 1) read
reg = reg + 1;          // 2) add 1
counter = reg;          // 3) write
```

If two threads perform these three steps at the same time, the steps can interleave. The following trace shows an order in which a counter that starts at 0 stays at 1 after two increments.

Step	Line	Call stack	Variables / state
1	1: `// counter++ is three CPU steps:`	—	counter=0, t1.reg=—, t2.reg=—
2	2: `int reg = counter; // 1) read`	t1	counter=0, t1.reg=0, t2.reg=—
3	2: `int reg = counter; // 1) read`	t2	counter=0, t1.reg=0, t2.reg=0
4	3: `reg = reg + 1; // 2) add 1`	t1	counter=0, t1.reg=1, t2.reg=0
5	4: `counter = reg; // 3) write`	t1	counter=1, t1.reg=1, t2.reg=0
6	3: `reg = reg + 1; // 2) add 1`	t2	counter=1, t1.reg=1, t2.reg=1
7	4: `counter = reg; // 3) write`	t2	counter=1, t1.reg=1, t2.reg=1

Table 10.9: One possible interleaving of the `counter++` steps. The "Call stack" column shows the thread running at that moment.

In step 2, `t1` reads the counter: its register gets 0. A context switch happens at exactly this point. In step 3, `t2` also reads the counter; because `t1` has not written yet, it also sees 0. In steps 4 and 5, `t1` adds 1 in its own register and writes 1 to memory.

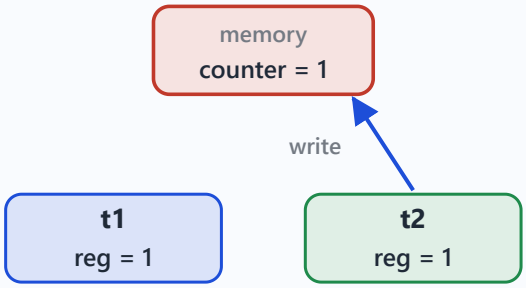


Figure 10.13: Step 7. `t2` added 1 to its own stale value and wrote it. Two increments were made, but `counter` is 1.

In steps 6 and 7, `t2` adds 1 to the old value in its own register, that is, to 0, and writes 1. A 1 has been written on top of the 1 that `t1` wrote. Two increments were made, but the result is 1: the increment of `t1` is lost (Figure 10.13). This event is called a **lost update**.

This is only one of the possible orders. If no interleaving occurs, that is, if one thread finishes its three steps before the other starts, the result is correct. In the program with 100,000 increments each, this interleaving happens thousands of times, and each time an increment is lost. The Thread Race simulator lets you order these steps by hand or at random and follow how the loss arises.

Example 10.4

The balance of an account is 500. `T1` calls `deposit(100)` and `T2` calls `withdraw(30)`; the methods consist only of the lines `balance += m` and `balance -= m` and are not synchronized. Suppose the order is: `T1` reads, `T2` reads, `T2` subtracts and writes, `T1` adds and writes. What is the balance in the end, and what should it have been?

The correct result is 500 + 100 − 30 = 570. `T1` reads 500. `T2` also reads 500, subtracts 30, and writes 470. `T1` adds 100 to the stale 500 and writes 600. The result is 600: the withdrawal by `T2` is lost. If the last two writes had happened in the opposite order, the result would be 470, and this time the deposit would be lost. The solution is to protect both methods with the lock of the same object (`synchronized`).

10.6.4 Synchronization options

Java offers many tools for coordinating threads. Each of them solves a different problem:

- `join()` : waiting for a thread to finish.
- `synchronized` : mutual exclusion with an object's monitor lock.
- `Lock` **objects** (`ReentrantLock` , `ReadWriteLock`): explicit locking.
- **Barrier, semaphore, exchanger, countdown latch**: `CyclicBarrier` , `Semaphore` , `Exchanger` , `CountDownLatch` .
- `volatile` **variables**: making reads and writes visible to all threads.
- **Atomic objects**: indivisible operations such as those of `AtomicInteger` .

There are also tools that look like synchronization but are not: priorities, putting a thread to sleep with `sleep` , giving way with `yield` , and timers. They do not guarantee order. Code written with the thought "I'll wait a bit, the other one will have finished" will one day produce a wrong result at a moment when the other one has not finished.

10.6.5 `synchronized` and monitors

Definition 10.7

A **monitor** is an internal lock and waiting mechanism attached to every Java object that can be locked and unlocked. Only one thread can hold a monitor at a time. The `synchronized` keyword acquires the monitor of an object; when the block ends, the monitor is released automatically.

The monitor works like this. While `t1` holds an object's monitor, if `t2` also wants the same monitor, `t2` waits; its state is `BLOCKED`. When the monitor is released, the waiting threads compete and only one of them gets the monitor. The monitor is released both when the block ends normally and when it is left through an exception. In this way **mutual exclusion** is achieved and data integrity is preserved.

Definition 10.8

A **critical section** is a piece of code that accesses shared data and must be executed by only one thread at a time. **Mutual exclusion** means that only one thread is in the critical section at a time.

The syntax is short. The object whose monitor will be used is written in parentheses:

```
synchronized (o) {
    // we hold o's monitor here
}
```

The counter increment in the race example is protected by such a block:

```
Runnable r = () -> {
    for (int i = 0; i < 50_000; i++) {
        synchronized (o) {   // take o's monitor
            shared++;
        }                    // monitor released
    }
};
```

When two threads run this task 50,000 times each, the result is 100000 every time. The program was run 10 times and always gave this value. Every `shared++` is done while the monitor is held; the other thread cannot enter the block until one thread has finished its read–add–write steps.

The operation of the monitor goes step by step like this:

```
// t1 and t2 run the same code
synchronized (o) {
    shared++;     // critical section
}
```

1. `t1` and `t2` reach the line `synchronized (o)` at the same time; the monitor is free.
2. `t1` wins and acquires the monitor. `t2` waits in the entry set, `BLOCKED`.
3. `t1` leaves the block; the monitor is released automatically. The waiting threads compete.

4. `t2` acquires the monitor and enters the critical section.
5. `t2` leaves too. `shared++` has been done twice, without interleaving.

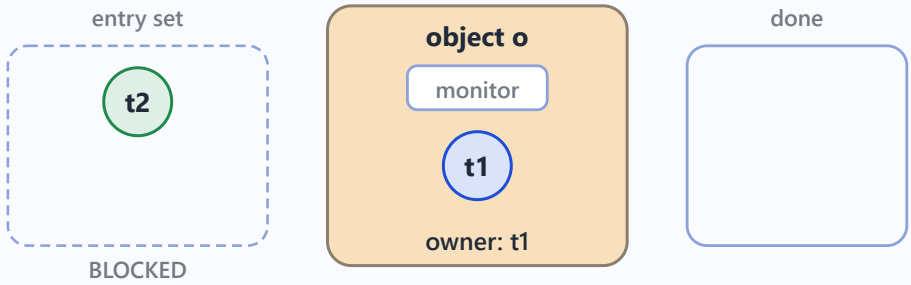


Figure 10.14: Step 2. `t1` owns the monitor; `t2` waits in the entry set, `BLOCKED`.

Which thread enters first is not determined. What matters is that the critical section is never executed by two threads at the same time. For monitor locking to work correctly, all threads must lock **the same object**. If each thread takes the monitor of its own object, there is no protection at all.

10.6.6 Synchronized methods

`synchronized` can also be written in front of a method. This is the same as wrapping the whole body of the method in a `synchronized` block.

```
class C {
    synchronized void p() { ... }
    static synchronized void q() { ... }
    //...
}
```

The equivalent of the definition above is:

```
class C {
    void p() { synchronized (this) { ... } }
    static void q() { synchronized (C.class) { ... } }
    //...
}
```

Which object is locked depends on the kind of method:

- **Instance method**: the monitor of the `this` object is locked.
- `static` **method**: since there is no instance, the monitor of the class's `C.class` object is locked.

For this reason, a `static synchronized` method and an instance `synchronized` method do not block each other; they use different locks. This behavior was tried on JDK 21: while another thread held the monitor of `C.class` for 500 ms, `x.p()` ran without waiting, whereas `C.q()` waited until the monitor was released. When a body is written in place of `...` in the code from the slides (for example `n++`, `s++`), both forms compile and behave the same.

The block form has an advantage: only the lines that really need it are locked. If only two lines of a long method touch shared data, locking the whole method makes the other threads wait for nothing.

Example 10.5

The class `Counter` implements `Runnable`; it has a `static int total` field, and its `run()` method is `synchronized`. `run()` increments `total` 100,000 times. Two threads run two separate objects with `new Thread(new Counter())`. Why can the total come out smaller than 200,000?

Each thread takes the monitor of its own object (`this`). Two different locks do not block each other, while the data they are supposed to protect is the shared `static total`. The program was run ten times, and the totals found ranged from 111,143 to 169,427; none was 200,000. The fix is for everyone to use the same lock: the increment in the loop goes into a `synchronized (Counter.class)` block. The corrected program gave 200000 in all five runs. `AtomicInteger` is another solution.

10.6.7 Preventing the race

The counter in the race example is fixed by moving it into a class whose methods are synchronized.

```
class Counter {
    private int value = 0;
    public synchronized void increment() { value++; }
    public synchronized int getValue() { return value; }
}

public class SyncCounter {
    public static void main(String[] args)
            throws InterruptedException {
        Counter counter = new Counter();
        Runnable task = () -> {
            for (int i = 0; i < 100_000; i++) {
                counter.increment();
            }
        };
        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Result: " + counter.getValue());
    }
}
```

```
Result: 200000
```

`increment()` is now `synchronized`: the read–add–write steps are performed without being split. The program was run 10 times and printed `Result: 200000` every time. The reading method `getValue()` is synchronized too. At first sight this may look unnecessary, because reading an `int` is a single step anyway. But if the read is not synchronized, there is no guarantee that the reading thread sees the current value. What was written before a lock is released becomes visible to the thread that acquires the same lock afterward.

Synchronization has a cost. Threads enter the critical section one at a time and wait for each other; the program becomes somewhat slower. That is why only shared data is protected. Local variables that only a single thread uses need no lock.

10.6.8 Mutual exclusion: a priority queue

Synchronized methods are the shortest way to make a whole data structure thread-safe. The following priority queue returns the smallest element on each call. All methods use the same monitor, `this`.

```
class MinQueue {
    private int[] data;
    private int size;
    public MinQueue(int n) { data = new int[n]; }
    public synchronized void add(int o) {
        if (size == data.length)
            throw new HeapFullException();
        data[size++] = o;
    }
    public synchronized int remove() {   // the smallest
        int min = 0;
        for (int i = 1; i < size; i++) {
            if (data[i] < data[min]) min = i;
        }
        int result = data[min];
        data[min] = data[--size];
        return result;
    }
    public synchronized String toString() {
        return Arrays.toString(Arrays.copyOf(data, size));
    }
}
```

`HeapFullException` is not defined in the slides; here it was written as a class that extends `RuntimeException` and carries the message `"Queue full"`. The queue is used from a single thread as follows:

```
MinQueue q =
        new MinQueue(3);
q.add(5);
q.add(2);
q.add(8);
System.out.println(q);
int x = q.remove();
System.out.println(x);
System.out.println(q);
q.add(1);
try {
    q.add(9);
} catch (HeapFullException e) {
    String m = e.getMessage();
    System.out.println(m);
}
```

```
[5, 2, 8]
2
[5, 8]
Queue full
```

After 5, 2, and 8 are added, the array is `[5, 2, 8]`. `remove()` finds the position of the smallest element by linear search: the 2 at position 1. That element is returned, the last element (8) is moved into its place, and the size decreases by one: `[5, 8]`. When 1 is added, the queue is full again. `add(9)` throws an exception, and the message is printed.

Why is synchronization essential in this class? The line `data[size++] = o` in `add` also contains a read–modify–write. If two threads add at the same time, both may read the same `size` and write to the same slot; one element is lost. If an `add` slips in the middle of `remove`, the position of the smallest element can change. Because all methods use the same monitor, no other method can start while one is running. Making `toString()` synchronized as well prevents printing a half-modified array.

10.7 Visibility

10.7.1 `volatile` fields

A race condition arises when two threads write at the same time. There is a subtler problem too: a value written by one thread may never be seen by another. Let us describe it with an intuitive model. For speed, processors keep values in caches close to the core or in registers. A compiler may also read a variable it believes does not change in a loop just once and keep it in a register. In that case, even if a thread on another core writes the new value to main memory, the first thread keeps working with the old value.

Definition 10.9

In a field declared with the `volatile` keyword, every read sees the last write to that field: a `volatile` write happens-before every subsequent read of the same field. In the intuitive model, this can be thought of as the value being read from main memory on every read and made visible to all threads immediately on every write. `volatile` guarantees visibility; it does not make multi-step operations atomic.

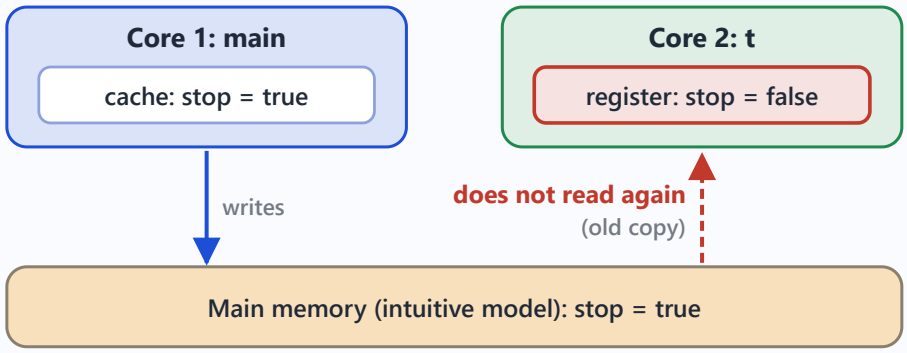


Figure 10.15: The intuitive model: `main` writes the new value to main memory; thread `t` keeps the old value in its register and does not see the change.

By the rules of the Java language, a single read or write of any variable other than `long` and `double` is already atomic. A 64-bit `long` or `double` value, however, may be written as two 32-bit halves on some JVMs. For `volatile long` and `volatile double` fields, this half-write cannot happen; reads and writes become atomic.

For a simple shared flag, `volatile` is enough; neither `synchronized` nor a lock is needed. But only a single read or a single write is atomic. Read–modify–write operations such as `x++` are not atomic on a `volatile` field either.

10.7.2 The visibility problem

```
public class VisibilityProblem {
    static boolean stop = false;   // NOT volatile!
    public static void main(String[] args)
            throws InterruptedException {
        Thread t = new Thread(() -> {
            long laps = 0;
            while (!stop) {
                laps++;
            }
            System.out.println("thread stopped");
        });
        t.setDaemon(true);
        t.start();
        Thread.sleep(500);
        stop = true;
        System.out.println("main: stop = true");
        t.join(2000);   // wait at most 2 s
        boolean alive = t.isAlive();
        System.out.println("still running? " + alive);
    }
}
```

The thread checks `stop` in an empty loop. After 500 ms, `main` sets `stop` to `true` and waits at most two seconds. Without `volatile`, the output in the slides is:

```
main: stop = true
still running? true
```

The thread never saw the change and is still running after two seconds. The cause is the JIT compiler. After the loop has run a few thousand times, the JIT translates it to machine code. Because nothing inside the loop changes `stop`, `stop` is read once and the loop runs like `while (true)`. This transformation is entirely correct for a single-threaded program, and the Java rules allow it for a field that is not `volatile`.

The program was run 10 times on JDK 21, and the thread failed to see the change every time; the output was like the one in the slides. When the field was declared `static volatile boolean stop`, the thread stopped in all ten runs:

```
main: stop = true
thread stopped
still running? false
```

In one of these runs, the line `thread stopped` was printed before `main: stop = true`. This is not a contradiction: `main` first changes the field and then prints; the thread can see the change between these two steps and print its own line first.

The outcome can vary with the JVM, the processor, and when the JIT kicks in. On one machine the problem may never show up, on another it shows up every time. Correctness is not left to this kind of luck: a flag changed by another thread is always declared `volatile`. That is also the purpose of the `setDaemon(true)` call in the example: the stuck thread must not prevent the program from exiting.

10.7.3 The limits of `volatile`

`volatile` does not solve the counter problem. Even if the counter in the race example is declared `volatile`, increments are lost:

```
static volatile int counter = 0;
// inside main:
Runnable r = () -> {
    for (int i = 0; i < 100_000; i++) counter++;
};
```

Three runs in the slides gave 163299, 164441, and 199332. The values found in 20 runs made for these notes:

```
157591, 188201, 171134, 181949, 163135, 175082, 150206, 177784, 184817, 187816, 178180,
174558, 168901, 171061, 179658, 173836, 168144, 176917, 170584, 172321
(smallest 150206, largest 188201)
```

The visibility problem is solved: every thread sees the current value on every read. But two threads can still read the same value and write the same result; the other thread can slip in between the read and the write. The solution is `synchronized` or `AtomicInteger`.

`volatile` cannot be combined with `final` either:

```
public class FinalVolatile {
    private final volatile int x = 5;
}
```

```
FinalVolatile.java:2: error: illegal combination of modifiers: final and volatile
    private final volatile int x = 5;
                               ^
1 error
```

The two keywords solve the same problem in different ways. Table 10.10 summarizes the difference.

`final`	`volatile`
Assigned once (in the constructor); never changes afterward	The value can change
When the constructor finishes (if `this` was not leaked from the constructor), everyone sees the same value	Every read returns the current value

Table 10.10: `final` and `volatile`.

A `final` field never changes, so there is no visibility problem; Java guarantees that after the constructor finishes, all threads will see the correct value of the `final` field, provided `this` was not leaked to another thread before the constructor finished. `volatile` is meaningful only for fields that change. Because writing both together is contradictory, the compiler rejects it.

10.7.4 The Java Memory Model and happens-before

When does one thread see a value written by another? The precise answer to this question is given by the Java Memory Model (JMM). The programmer does not have to think about processor caches and memory barriers; instead, they reason with happens-before relationships.

Definition 10.10

If operation A **happens-before** operation B, everything A wrote is guaranteed to be visible in B. The happens-before relation is transitive: if A happens-before B and B happens-before C, then A happens-before C.

Rule	A happens-before B
Lock	Releasing a monitor or lock → a later acquisition of the same lock
`volatile`	Writing a `volatile` field → a later read of the same field
Start	Everything before the call `t.start()` → the actions inside `t`
Termination	All actions of `t` → the thread that returns from `t.join()`
Ready-made structures	`BlockingQueue.put` → the `take` that receives the same element; the work of a `Future` → `get()`

Table 10.11: The main rules that establish a happens-before relation.

Visibility behavior in multithreaded programs is explained by these rules. In the `join()` example, `result` was not `volatile`, yet `main` saw the correct value: the termination rule. In the synchronized counter, `getValue()` being synchronized as well brings the lock rule into play. In the flag example, the `volatile` rule applies. If none of these relationships is established, there is no visibility guarantee between two threads.

There is also a performance problem that must not be confused with visibility: **false sharing**. Independent variables of different threads sit on the same cache line (usually 64 bytes), and if they are updated often, the cores keep pulling the line away from each other. The result is not wrong, but the program slows down. This is a performance problem, not a correctness problem.

10.8 Explicit locks and deadlock

10.8.1 The `Lock` interface

`synchronized` is simple and safe, but it is not flexible. A thread cannot give up while waiting for a lock, the wait cannot be limited by a timeout, and a waiting thread cannot be interrupted. For complex synchronization scenarios, the `java.util.concurrent.locks` package offers explicit locks.

Definition 10.11

An **explicit lock** is an object that implements the `Lock` interface. The lock is acquired with `lock()` and released with `unlock()`; the programmer calls both. `ReentrantLock` and `ReentrantReadWriteLock` are the main implementations of this interface.

Explicit locks do three things that `synchronized` cannot: try without waiting (`tryLock()`), wait for a limited time (`tryLock(time)`), and wait in an interruptible way (`lockInterruptibly()`). In return, the responsibility is the programmer's: the lock must not be forgotten. To use them, import `java.util.concurrent.locks.*`.

```
public class LockDemo {
    private final Lock lock = new ReentrantLock();
    private int counter = 0;
    public void work() {
        lock.lock();
        try {
            // critical section
            counter++;
        } finally {
            lock.unlock();
        }
    }
    public static void main(String[] args)
            throws InterruptedException {
        LockDemo o = new LockDemo();
        Runnable r = () -> {
            for (int i = 0; i < 100_000; i++) o.work();
        };
        Thread a = new Thread(r), b = new Thread(r);
        a.start(); b.start();
        a.join(); b.join();
        System.out.println("Result: " + o.counter);
    }
}
```

```
Result: 200000
```

The program is the race example rewritten with an explicit lock. Every `counter++` is done while the lock is held. The program was run 10 times and printed `Result: 200000` every time. The lock object is in a `final` field; all threads use the same lock.

10.8.2 Unlocking in a `finally` block

The correct usage pattern for an explicit lock is always the same:

```
public void work() {
    Lock lock = ...;
    lock.lock();
    try {
        // access the guarded resource
    } finally {
        lock.unlock();
    }
}
```

`lock()` is called right before the `try`. `unlock()` is inside the `finally`. Even if an exception occurs in the critical section, `finally` runs and the lock is released. There is also a reason why `lock()` is written before the `try` and not inside it. If `lock()` fails, we do not want `finally` to try to release a lock that was never acquired.

```
lock.lock();
doWork();       // if this throws...
lock.unlock();  // ...we never get here
```

In this wrong form, if `doWork()` throws an exception, the `unlock()` line is never reached. The lock stays locked, and every thread that wants it waits forever. `synchronized` has no such problem: even if the block is left through an exception, the monitor is released automatically.

Example 10.6

The method `divide(a, b)` acquires the lock, prints `a / b`, and releases the lock; there is no `finally`. `t1` calls `divide(10, 0)`; after `t1` finishes, `t2` calls `divide(10, 2)`. What is seen on the screen, and in what state does `t2` remain?

In `t1`, `10 / 0` throws `ArithmeticException`. `t1` terminates, but because `unlock()` is never reached, the lock stays locked. Only the stack trace of `t1` appears on the screen; `5` is never printed. `t2` waits in `lock()` forever. The test program was made to finish without hanging by turning `t2` into a daemon thread and printing its state after 300 ms; in all three runs the state was `WAITING`. Because waiting on a `Lock` is not a monitor wait, the state is `WAITING`, not `BLOCKED`. When `unlock()` was moved into `finally`, `t2` printed `5` and finished.

A second error is releasing a lock that was never acquired:

```
Lock lock = new ReentrantLock();
lock.unlock();   // never acquired
```

```
Exception in thread "main" java.lang.IllegalMonitorStateException
    at
java.base/java.util.concurrent.locks.ReentrantLock$Sync.tryRelease(ReentrantLock.java:175)
    at
java.base/java.util.concurrent.locks.AbstractQueuedSynchronizer.release(AbstractQueuedSync
hronizer.java:1059)
    at java.base/java.util.concurrent.locks.ReentrantLock.unlock(ReentrantLock.java:494)
    at UnlockError.main(UnlockError.java:5)
```

`ReentrantLock` keeps track of the owner of the lock. If a thread that is not the owner calls `unlock()`, `IllegalMonitorStateException` is thrown. The full message printed by JDK 21 also shows where the exception occurred.

10.8.3 `tryLock()`

`tryLock()` tries to acquire the lock without waiting. If the lock is free, it takes it and returns `true`; if it is held, it returns `false` without waiting at all. The form `tryLock(3, TimeUnit.SECONDS)` waits at most three seconds. Because the timed wait can be interrupted, `InterruptedException` must be declared.

```
static void work()
        throws InterruptedException {
    if (lock.tryLock()) {
        // got it
        try { System.out.println("1: acquired"); }
        finally { lock.unlock(); }
    } else {
        // didn't get it; do something else
        System.out.println("1: not acquired");
    }
    if (lock.tryLock(3, TimeUnit.SECONDS)) {
        try { System.out.println("2: acquired"); }
        finally { lock.unlock(); }
    } else {
        // not acquired within 3 seconds
        System.out.println("2: not acquired");
    }
}
```

The method is tested as follows: `main` acquires the lock, starts `work()` in a new thread, and releases the lock one second later.

```
lock.lock();   // main holds the lock
Thread t = new Thread(() -> {
    try { work(); }
    catch (InterruptedException e) { }
});
t.start();
Thread.sleep(1000);
System.out.println("main: releasing the lock");
lock.unlock();
t.join();
```

```
1: not acquired
main: releasing the lock
2: acquired
```

On the first attempt the lock is held by `main`; `tryLock()` returns `false` without waiting, and `1: not acquired` is printed. The thread then starts waiting for at most three seconds. One second later, `main` releases the lock; the waiting thread acquires it immediately and prints `2: acquired`. The output was the same in all five runs. Had `main` never released the lock, the second attempt would have returned `false` after three seconds; this case was tried separately and `2: not acquired (3 s)` was seen.

10.8.4 `ReentrantLock` and reentrancy

The name `ReentrantLock` means "a lock that allows re-entry". If the thread that holds the lock asks for the same lock once more, it does not wait; only a counter increases. The lock is really released when the counter drops back to 0.

Definition 10.12

A **reentrant lock** is a lock that lets the thread holding it acquire it again without waiting. The lock stores an owner and a **hold count**; each `lock()` increases the count by one, and each `unlock()` decreases it by one.

Nested critical sections with more than one lock can also be entered:

```
ReentrantLock lock1 = new ReentrantLock();
// true: a fair lock
ReentrantLock lock2 = new ReentrantLock(true);
// ...
lock1.lock();
try {
    // enter the critical section
    lock2.lock();
    try {
        // enter the nested section
        System.out.println("both locks acquired");
    } finally {  lock2.unlock();  }
} finally {  lock1.unlock();  }
System.out.println("fair? " + lock2.isFair());
```

```
both locks acquired
fair? true
```

`new ReentrantLock(true)` creates a **fair lock**: when the lock becomes free, it is given to the thread that has waited longest. The no-argument constructor gives a non-fair lock; `lock1.isFair()` returns `false`. A non-fair lock is faster, but it may let a thread keep waiting for a long time. When two locks are acquired in a nested way, the release order is the reverse of the acquisition order: first `lock2`, then `lock1`.

`synchronized` is reentrant too. A `synchronized` method can call another `synchronized` method of the same object; the thread does not wait for a monitor it already holds. Without reentrancy, such a call would lock the thread against itself.

The following program shows how the hold count changes.

```
import java.util.concurrent.locks.ReentrantLock;

public class Reentrancy {
    static final ReentrantLock lock =
            new ReentrantLock();
    static void show(String where) {
        System.out.println(where + lock.getHoldCount());
    }
    static void outer() {
        lock.lock();
        try { show("outer(): "); inner(); }
        finally { lock.unlock(); }
    }
    static void inner() {
        lock.lock();   // enters without waiting
        try { show("inner(): "); }
        finally { lock.unlock(); }
    }
    public static void main(String[] args) {
        outer();
        show("main:    ");
    }
}
```

Step	Line	Call stack	Variables / state	New output
1	20: `outer();`	main()	owner=none, count=0	—
2	10: `lock.lock();`	main() → outer()	owner=main, count=1	—
3	11: `try { show("outer(): "); inner(); }`	main() → outer() → show()	owner=main, count=1	`outer(): 1`
4	15: `lock.lock(); // enters without waiting`	main() → outer() → inner()	owner=main, count=2	—
5	16: `try { show("inner(): "); }`	main() → outer() → inner() → show()	owner=main, count=2	`inner(): 2`
6	17: `finally { lock.unlock(); }`	main() → outer() → inner()	owner=main, count=1	—
7	12: `finally { lock.unlock(); }`	main() → outer()	owner=none, count=0	—
8	21: `show("main: ");`	main() → show()	owner=none, count=0	`main: 0`

Table 10.12: Trace of the reentrancy program.

In step 2, `outer()` acquires the lock: owner `main`, hold count 1. In step 4, `inner()` asks for the same lock. Because the owner of the lock is already `main`, it does not wait; the count becomes 2.

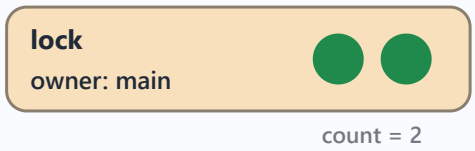


Figure 10.16: Step 4. The owner of the lock is `main`, and the hold count is 2.

In step 6, the `finally` of `inner()` calls `unlock()`. The count drops to 1; the lock is still held by `main` and no other thread can acquire it. In step 7, the `finally` of `outer()` brings the count down to 0, and the lock is really released. The last `show` call in `main` therefore prints 0. Because the output comes from a single thread, it is fixed. The rule is clear: every `lock()` needs one `unlock()`. An extra `unlock()` throws `IllegalMonitorStateException`; a missing `unlock()` leaves the lock locked forever.

10.8.5 `ReadWriteLock`

Many data structures are read often and written rarely: a settings table, a cache, a dictionary. Because reads do not change the data, there is no need for them to block each other. `ReadWriteLock` handles this situation with two separate locks.

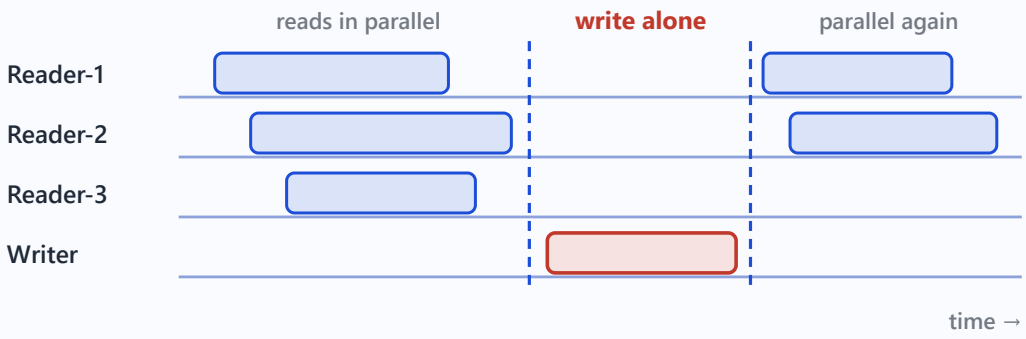


Figure 10.17: `ReadWriteLock`: readers read at the same time; nobody can read while the writer writes.

Read lock
: More than one thread can read the data at the same time; they do not block each other while reading.

Write lock
: While one thread writes, both reading and writing by the others are blocked; the write is safe and atomic.

```
private final Map<String, String> table =
        new HashMap<>();
private final ReadWriteLock lock =
        new ReentrantReadWriteLock();

public String get(String key) {
    lock.readLock().lock();   // lock for reading
    try {
        return table.get(key);   // read the data
    } finally {
        lock.readLock().unlock();
    }
}
public void put(String key, String value) {
    lock.writeLock().lock();  // lock for writing
    try {
        table.put(key, value);   // write the data
    } finally {
        lock.writeLock().unlock();
    }
}
```

`ReentrantReadWriteLock` is an implementation of the `ReadWriteLock` interface. `get` uses the read lock and `put` uses the write lock. The code that tries the class with two readers is:

```
Settings s = new Settings();
s.put("lang", "en");
Runnable reader = () -> System.out.println(
        Thread.currentThread().getName()
        + ": " + s.get("lang"));
Thread r1 = new Thread(reader, "Reader-1");
Thread r2 = new Thread(reader, "Reader-2");
r1.start(); r2.start();
r1.join(); r2.join();
```

```
Reader-2: en
Reader-1: en
```

Two readers can hold the read lock at the same time. This was tested separately: while one thread held the read lock, a call to `readLock().tryLock()` from a second thread returned `true` and `writeLock().tryLock()` returned `false`. The order of the lines varies: in 40 runs, `Reader-1` printed first 22 times and `Reader-2` 18 times.

10.8.6 Explicit versus intrinsic locking

	Explicit: `Lock`	**Intrinsic:** `synchronized`
Control	The programmer calls `lock()` / `unlock()`	Automatic: acquired on entering the block, released on leaving it
Classes	The `Lock` interface, `ReentrantLock`, `ReentrantReadWriteLock`	A keyword; the monitor of every object
Trying without waiting	`tryLock()`, `tryLock(time)`	None: waits until the lock is free
Interruptible waiting	`lockInterruptibly()`	None
Fairness, conditions	`new ReentrantLock(true)`, several `Condition`s	None; a single wait set
Reentrancy	`ReentrantLock` is reentrant	Always reentrant
Risk	If `unlock()` is forgotten, the lock stays held forever	No risk of forgetting
When to use	When advanced control is needed	Simple scenarios, more readable code

Table 10.13: Explicit (`Lock`) and intrinsic (`synchronized`) locking.

Table 10.13 makes the choice easier. The default choice is `synchronized`: it is short, readable, and hard to get wrong. `Lock` is used when one of the extra capabilities in the table is really needed.

10.8.7 Deadlock

Definition 10.13

A **deadlock** occurs when two or more threads wait forever for locks held by one another. It arises when locks are acquired in different orders by different threads.

```
static void useBoth(Object x, Object y, String name) {
    synchronized (x) {        // Lock 1
        pause(100);
        synchronized (y) {    // Lock 2
            System.out.println(name + " done");
        }
    }
}
public static void main(String[] args)
        throws InterruptedException {
    Thread t1 = new Thread(() -> useBoth(a, b, "T1"));
    Thread t2 = new Thread(() -> useBoth(b, a, "T2"));
    t1.setDaemon(true);
    t2.setDaemon(true);
    t1.start();
    t2.start();
    Thread.sleep(500);
    String s = t1.getState() + " " + t2.getState();
    System.out.println(s);
    long[] ids = ManagementFactory.getThreadMXBean()
            .findDeadlockedThreads();
    int n = (ids == null) ? 0 : ids.length;
    System.out.println("Deadlocked: " + n);
}
```

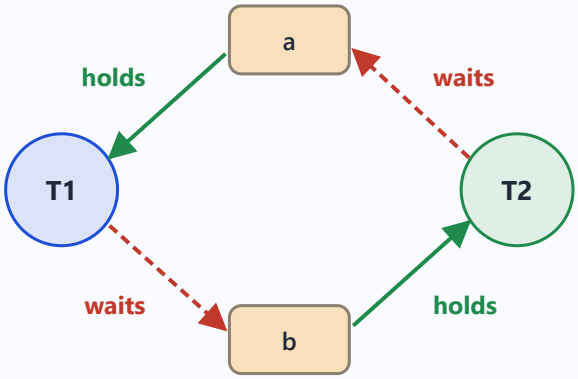


Figure 10.18: A deadlock cycle: T1 holds `a` and waits for `b`; T2 holds `b` and waits for `a`.

```
BLOCKED BLOCKED
Deadlocked: 2
```

`useBoth(x, y, name)` first acquires the monitor of `x` and then that of `y`. T1 calls `useBoth(a, b)`, while T2 calls `useBoth(b, a)` in the opposite order. `pause(ms)` is a helper that wraps `sleep` in `try`-`catch`; it makes both threads wait after acquiring their first lock and before asking for the

second. After 500 ms, `main` prints the states of the two threads: both are `BLOCKED`. `ThreadMXBean.findDeadlockedThreads()` returns the IDs of the deadlocked threads; the count is 2.

The two threads were made daemon threads. This way, when `main` ends the JVM exits, and the deadlock does not leave the program hanging. The program was run 10 times, and `BLOCKED BLOCKED` and `Deadlocked: 2` were seen every time. Because of the 100 ms sleep, the two threads almost always acquire their first locks together. Without the sleep, the deadlock would occur only in some runs.

Let us follow step by step how the deadlock is set up, using the shorter version from the slides. In this version the threads are not daemon threads; the program really waits forever.

```
static final Object a = new Object(), b = new Object();
static void useBoth(Object x, Object y, String name) {
    synchronized (x) {
        pause(100);
        synchronized (y) {
            System.out.println(name + " done");
        }
    }
}
public static void main(String[] args) {
    new Thread(() -> useBoth(a, b, "T1")).start();
    new Thread(() -> useBoth(b, a, "T2")).start();
}
```

Step	Line	Call stack	Variables / state
1	1: `static final Object a = new Object(), b = new Object();` 10: `public static void main(String[] args) {`	main()	a=Object, b=Object
2	11: `new Thread(() -> useBoth(a, b, "T1")).start();` 3: `synchronized (x) {`	T1: useBoth(a, b)	T1: x=a, y=b
3	12: `new Thread(() -> useBoth(b, a, "T2")).start();` 3: `synchronized (x) {`	T2: useBoth(b, a)	T2: x=b, y=a
4	4: `pause(100);`	T1: pause(100) → T2: pause(100)	a=held by T1, b=held by T2
5	5: `synchronized (y) {`	T1: useBoth(a, b)	T1 wants=b
6	5: `synchronized (y) {`	T1: useBoth(a, b) → T2: useBoth(b, a)	T2 wants=a
7	6: `System.out.println(name + " done");`	T1: useBoth(a, b) → T2: useBoth(b, a)	T1=BLOCKED, T2=BLOCKED

Step	Line	Call stack	Variables / state
8	12: `new Thread(() -> useBoth(b, a, "T2")).start();`	T1: useBoth(a, b) → T2: useBoth(a, b)	T2 call=useBoth(a, b, "T2")

Table 10.14: How the deadlock forms, step by step.

In steps 2 and 3, T1 acquires `a` and T2 acquires `b`. So far there is no problem. In step 4, both sleep for 100 ms; `sleep` does not release a held lock. In step 5, T1 wakes up and asks for `b` for the inner block. `b` is held by T2: T1 becomes `BLOCKED`. In step 6, T2 also asks for `a`; `a` is held by T1. Each waits for the other to release, and a cycle forms.

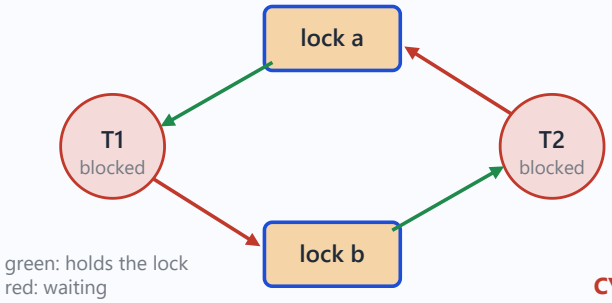


Figure 10.19: Step 6. The cycle is closed in the resource allocation graph: T1 → `b` → T2 → `a` → T1.

Neither can ever reach the `"done"` line. For these notes this program was run three times with a 6-second timeout. In all three, no line was printed and the process was terminated from outside at the timeout. A deadlock is never resolved without outside intervention; the JVM does not notice it and fix it on its own.

10.8.8 Preventing deadlock

The most effective remedy for deadlock is a single change: all threads acquire locks in the same order.

```
Thread t1 = new Thread(() -> useBoth(a, b, "T1"));
Thread t2 = new Thread(() -> useBoth(a, b, "T2"));
```

```
T1 done
T2 done
TERMINATED TERMINATED
Deadlocked: 0
```

Everyone acquires `a` first and then `b`. T2 waits for `a` and never holds `b`; no cycle can form. When T1 finishes, it releases the locks, T2 acquires them, and it finishes too. The program was run 10 times, and the output was these four lines every time.

There are four ways to prevent deadlock:

Fixed lock order
: All threads acquire locks in the same order; for example, by the ID number of the objects.

Timed attempt
: If `tryLock(time)` cannot acquire the lock, the thread releases the locks it holds and tries again later.

Short critical sections
: Do not ask for another lock while holding one, and do not do long work while holding a lock.

Diagnosis
: The JDK's `jstack` tool or `ThreadMXBean.findDeadlockedThreads()` shows the deadlocked threads and the locks they hold.

Common mistake

Ignoring deadlock because "it never happens anyway". In a version without the sleep, if T1 takes `a` then `b` and T2 takes `b` then `a`, in most runs one of them acquires both locks and leaves, and the problem is not visible. For these notes, two threads each ran these two blocks 1,000 times, and the experiment was repeated at most 200 times. In all three separate runs, the deadlock occurred before this limit was reached. When a deadlock shows up varies from run to run; if its probability is not zero, it will happen sooner or later.

10.8.9 Livelock and starvation

Deadlock is not the only progress problem. It has two relatives.



Figure 10.20: Deadlock, livelock, and starvation.

Livelock
: The threads are not blocked; they keep changing state by constantly "giving way" to each other, but none of them can finish its work. It is like two people meeting in a narrow corridor who step to the same side at the same time. Example: two threads release the lock they hold when `tryLock` fails, try again at the same moment, and always collide.

Starvation
: A thread cannot reach a resource for a long time, or at all: others always grab the lock, or long critical sections keep it waiting. Priority values (`setPriority`) are not a reliable way to solve this problem.

```
Lock fair = new ReentrantLock(true);  // fair
if (lock.tryLock(50, TimeUnit.MILLISECONDS)) {
    try { /* work */ }
    finally { lock.unlock(); }
} else {
    Thread.sleep(rnd.nextInt(20)); // random
}
```

The remedies are short. A fair lock gives the lock to the thread that has waited longest; the price is slightly slower operation. Waiting a random time before retrying breaks the symmetry of the two threads and prevents livelock; in the code, `rnd.nextInt(20)` does this job. Keeping critical sections short also ensures that everyone gets a turn.

10.9 Condition synchronization

10.9.1 Waiting for a condition

Mutual exclusion prevents two threads from entering the critical section at the same time. In some cases that is not enough: when a thread enters the critical section, the condition suitable for its work may not yet hold. Consider a shared queue. A consumer takes items from the queue, but if the queue is empty, there is nothing to take. A producer adds items, but if the queue is full, there is no room.

Definition 10.14

Condition synchronization ensures that a shared resource is accessed only when a certain condition holds. A thread waits until the condition holds; the thread that changes the condition wakes the waiting one.

Updating the queue safely requires the object's lock. It is also necessary to wait until the condition holds. But one cannot wait while holding the lock. If a consumer holds the lock while waiting for an item to arrive in the queue, the producer can never acquire the lock and cannot add an item; everyone gets stuck.

The first solution that comes to mind is to check the condition continuously in a loop:

```
while (queue.isEmpty()) {
    // burns CPU; if the lock is held,
    // the producer can never add
}
```

This **busy waiting** is wrong in two ways. The loop keeps one core completely busy for nothing. And if the lock is held, the producer can never add, and the loop goes on forever. Java offers three correct solutions:

1. `Object.wait()` and `Object.notify()` / `notifyAll()`.
2. The `Condition` interface, used together with `Lock`.
3. Synchronizer objects (`CountDownLatch`, `Semaphore`, ...) and the ready-made `BlockingQueue` classes.

10.9.2 `wait()` and `notify()`

Two sets are attached to the monitor of every object. The entry set holds the threads that want to acquire the monitor. The **wait set** holds the threads that are waiting for a condition to change.

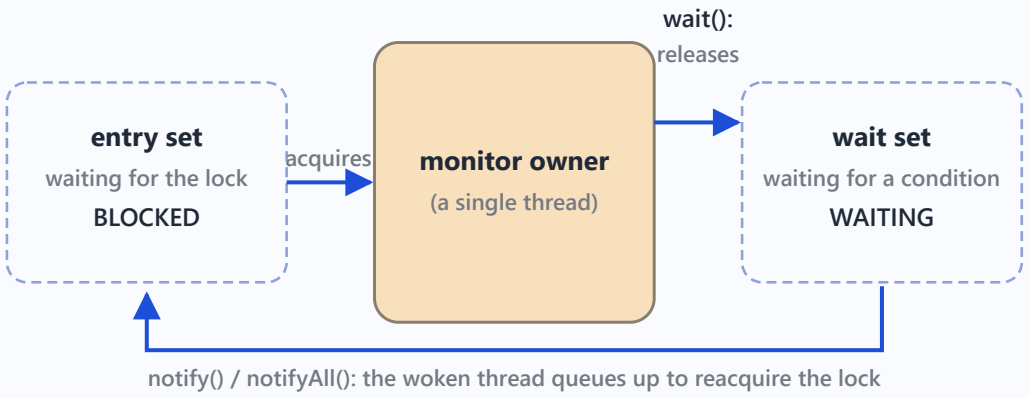


Figure 10.21: The monitor: entry set, owner, and wait set. `wait` moves a thread to the wait set, and `notify` returns it to the entry set.

- The thread that holds the lock calls `wait()`: it enters the wait set, releases the lock, and blocks.
- `notify()` wakes one thread from the wait set, and `notifyAll()` wakes all of them. The form `wait(ms)` also returns when the time is up.
- The woken thread must first reacquire the lock. Once it has the lock, it returns from `wait()` and continues where it left off.

There are two rules. First, `wait` and `notify` are called only inside a `synchronized` block or method of the same object. Otherwise JDK 21 throws `java.lang.IllegalMonitorStateException: current thread is not owner`. Second, the condition is checked with `while`, not `if`. By the time the woken thread reacquires the lock, another thread may have invalidated the condition again. Java also allows a thread to wake up without any `notify`, a **spurious wakeup**. The `while` loop rechecks the condition on every wakeup.

Example 10.7

Inside a `synchronized` block, what is the difference between `lock.wait()` and `Thread.sleep(1000)`?

`wait()` releases the monitor and enters the wait set; when it is woken by `notify` or `notifyAll`, it reacquires the lock and continues. `sleep()` has nothing to do with the lock: it sleeps while holding it, and during that time everyone who wants the lock waits `BLOCKED`. Holding the lock while waiting for a condition keeps out the thread that would change the condition; that is why `wait()` is used for waiting on a condition.

10.9.3 Blocking queues

The classic example of condition synchronization is the **producer–consumer problem**. Producer threads put items into a shared buffer, and consumer threads take them. If the buffer is full, the producer waits; if it is empty, the consumer waits. The following `BlockingQueue` class implements this buffer with a fixed-size array.

```
class BlockingQueue {
    private Object[] data;
    private int head = 0;
    private int tail = 0;
    private int count = 0;
    public BlockingQueue(int capacity) {
        if (capacity <= 0)
            throw new IllegalArgumentException(
                    "capacity must be > 0");
        data = new Object[capacity];
    }
    public synchronized int getSize() {
        return count;
    }
    public synchronized void add(Object item)
            throws InterruptedException {
        while (count == data.length) {
            wait();
        }
        data[tail] = item;
        tail = (tail + 1) % data.length;
        count++;
        notifyAll();
    }
```

```
    public synchronized Object remove()
            throws InterruptedException {
        while (count == 0) {
            wait();
        }
        Object item = data[head];
        data[head] = null;
        head = (head + 1) % data.length;
        count--;
        notifyAll();
        return item;
    }
}
```

The array is used circularly. `head` is the position of the element to be removed, and `tail` is the position where the next element will be added. On every add and remove the position advances by one; `% data.length` wraps it around at the end. `count` is the number of elements in the queue. `remove` sets the slot it read to `null`; this way the queue no longer holds an object that does not belong to it, and the garbage collector can reclaim it. The constructor rejects a capacity of 0 or less with `IllegalArgumentException`; in a queue of capacity 0, `add` would wait forever.

`add` waits when the queue is full (`count == data.length`); `remove` waits when the queue is empty. After every add or remove, `notifyAll()` wakes the waiting threads. Here `notifyAll()` is a deliberate choice. Both producers and consumers can be waiting in the same wait set. If `notify()` woke the wrong kind of thread, that thread would find the condition unsatisfied and wait again, and the right thread might never wake up.

10.9.4 Code trace: producer and consumer

Let us trace one possible order in a queue of capacity 2, in which the producer adds 1, 2, 3 and the consumer takes them. The line numbers are those of the two methods below.

```
public synchronized void add(Object item)
        throws InterruptedException {
    while (count == data.length) {
        wait();
    }
    data[tail] = item;
    tail = (tail + 1) % data.length;
    count++;
    notifyAll();
}
public synchronized Object remove()
        throws InterruptedException {
    while (count == 0) {
        wait();
    }
    Object item = data[head];
    data[head] = null;
    head = (head + 1) % data.length;
    count--;
    notifyAll();
    return item;
}
```

Step	Line	Variables / state
1	6: `data[tail] = item;` 7: `tail = (tail + 1) % data.length;` 8: `count++;`	head=0, tail=1, count=1
2	6: `data[tail] = item;` 7: `tail = (tail + 1) % data.length;` 8: `count++;`	head=0, tail=0, count=2
3	4: `wait();`	head=0, tail=0, count=2
4	16: `Object item = data[head];` 17: `data[head] = null;` 18: `head = (head + 1) % data.length;` 19: `count--;`	head=1, tail=0, count=1
5	20: `notifyAll();`	head=1, tail=0, count=1
6	3: `while (count == data.length) {` 6: `data[tail] = item;` 7: `tail = (tail + 1) % data.length;` 8: `count++;`	head=1, tail=1, count=2
7	16: `Object item = data[head];` 17: `data[head] = null;` 18: `head = (head + 1) % data.length;` 19: `count--;`	head=0, tail=1, count=1
8	16: `Object item = data[head];` 17: `data[head] = null;` 18: `head = (head + 1) % data.length;` 19: `count--;`	head=1, tail=1, count=0

Step	Line	Variables / state
9	14: `wait();`	head=1, tail=1, count=0

Table 10.15: Producer–consumer trace (capacity 2, one possible order).

In step 1, `add(1)` finds room: 1 is written to `data[0]` and `tail` advances to 1. In step 2, `add(2)` writes 2 to `data[1]`; `tail` wraps around to 2 % 2 = 0. The queue is full. In step 3, `add(3)` meets a full queue and calls `wait()`. The producer enters the wait set and releases the lock.

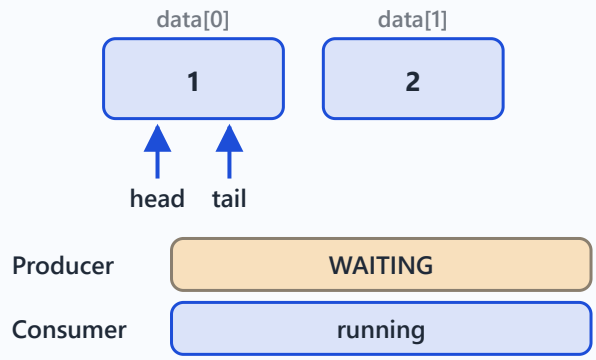


Figure 10.22: Step 3. The queue is full; the producer is in the wait set because of `wait()`, and the lock is free.

Because the lock is free, the consumer can enter in step 4. `remove()` takes 1, sets the slot to `null`, and `head` becomes 1. In step 5, `notifyAll()` wakes the producer. But the lock is still held by the consumer, so the producer waits for the lock. When `remove` returns 1 and releases the lock, the producer acquires it in step 6. It rechecks the `while` condition: the queue is no longer full. 3 is written to `data[0]`. In steps 7 and 8, the consumer takes 2 and 3; the queue becomes empty. In step 9, the next `remove()` meets an empty queue and the consumer waits. It will wake up when the producer adds 4 and calls `notifyAll()`.

This is one of the possible orders; the real order depends on the scheduler. But in no order does the queue overflow or get read while empty. `while` together with `wait` provides this guarantee.

The test program in the slides uses the queue with two threads. The consumer sleeps 100 ms before each item, so the producer fills the queue and has to wait.

```
BlockingQueue queue = new BlockingQueue(2);
Thread producer = new Thread(() -> {
    try {
        for (int i = 1; i <= 4; i++) {
            queue.add(i);
            System.out.println("produced: " + i);
        }
    } catch (InterruptedException e) { }
});
Thread consumer = new Thread(() -> {
    try {
        for (int i = 1; i <= 4; i++) {
            Thread.sleep(100);
            Object x = queue.remove();
            System.out.println("consumed: " + x);
        }
    } catch (InterruptedException e) { }
});
producer.start();
consumer.start();
producer.join();
consumer.join();
System.out.println("left: " + queue.getSize());
```

```
produced: 1
produced: 2
produced: 3
consumed: 1
produced: 4
consumed: 2
consumed: 3
consumed: 4
left: 0
```

In this output from the slides, `produced: 3` appears before `consumed: 1`. At first sight this is odd: 3 can be added only after 1 has been removed. The explanation is that `println` is outside the lock. The consumer returns from `remove` and is just about to print when the scheduler runs the producer; the producer adds 3 and prints its own line first. The program was run 40 times; the orders seen:

```
20 times:  produced: 1 · produced: 2 · produced: 3 · consumed: 1 · consumed: 2 · produced:
4 · consumed: 3 · consumed: 4
20 times:  produced: 1 · produced: 2 · produced: 3 · consumed: 1 · produced: 4 · consumed:
2 · consumed: 3 · consumed: 4
```

In all runs, the last line was `left: 0`, and every item was produced and consumed exactly once. The order of the lines varied from run to run.

10.9.5 The `Condition` interface

`wait` and `notify` belong to the world of monitors. In a class that uses `Lock`, the `Condition` interface does the same job.

> **Definition 10.15**
>
> A `Condition` is a wait set attached to a `Lock`; it is created with `lock.newCondition()`. `await()` releases the lock and suspends the thread; `signal()` wakes one waiting thread, and `signalAll()` wakes all of them.

The main advantage of `Condition` is that several conditions can be attached to one lock. Producers and consumers wait in separate sets; each thread wakes only the relevant one.

```
class BlockingQueue {
    private final Lock lock = new ReentrantLock();
    private final Condition notFull =
            lock.newCondition();
    private final Condition notEmpty =
            lock.newCondition();
    private Object[] data;
    private int head = 0;
    private int tail = 0;
    private int count = 0;
    public BlockingQueue(int capacity) {
        if (capacity <= 0)
            throw new IllegalArgumentException(
                    "capacity must be > 0");
        data = new Object[capacity];
    }
    // add() and remove()
}
```

```
public void add(Object item)
        throws InterruptedException {
    lock.lock();
    try {
        while (count == data.length)
            notFull.await();
        data[tail] = item;
        tail = (tail + 1) % data.length;
        count++;
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}
```

```
public Object remove()
        throws InterruptedException {
    lock.lock();
    try {
        while (count == 0)
            notEmpty.await();
        Object item = data[head];
        data[head] = null;
        head = (head + 1) % data.length;
        count--;
        notFull.signal();
        return item;
    } finally {
        lock.unlock();
    }
}
```

The thread that adds signals only the `notEmpty` condition, and the thread that removes signals only `notFull`. Only consumers wait on `notEmpty`, and only producers wait on `notFull`. Because there are no unnecessary wakeups, `signal()` is enough in place of `signalAll()`. The lock is released in `finally`; `await`, like `wait`, sits inside a `while` loop.

The slides test the class with a producer and `main`: the producer adds 1 through 4, and `main` removes four times and prints the items separated by spaces.

```
1 2 3 4
```

Because the queue works on the first-in, first-out (FIFO) principle, this output is always the same as far as order goes. The program was run ten times and always printed `1 2 3 4`.

10.9.6 A ready-made solution: `ArrayBlockingQueue`

Writing our own blocking queue is useful for understanding the mechanism. In real projects, the ready-made `BlockingQueue` implementations in the `java.util.concurrent` package are used: `ArrayBlockingQueue`, `LinkedBlockingQueue`, and others.

```
BlockingQueue<Integer> queue =
        new ArrayBlockingQueue<>(2);
Thread producer = new Thread(() -> {
    try {
        // put(): waits if the queue is full
        for (int i = 1; i <= 4; i++) queue.put(i);
    } catch (InterruptedException e) { }
});
producer.start();
for (int i = 1; i <= 4; i++) {
    // take(): waits if the queue is empty
    System.out.print(queue.take() + " ");
}
System.out.println();
producer.join();
```

```
1 2 3 4
```

`put()` waits if the queue is full, and `take()` waits if it is empty. The waiting and waking are done inside the class; the programmer writes no locks or conditions. The result was `1 2 3 4` in all ten runs. Table 10.16 compares the three ways.

Way	Waiting	Waking
`synchronized`	`wait()`	`notify()`, `notifyAll()`
`Lock` + `Condition`	`await()`	`signal()`, `signalAll()`
`ArrayBlockingQueue`	inside `put()`, `take()`	automatic

Table 10.16: Three ways to wait for a condition.

Example 10.8

With an `ArrayBlockingQueue<Integer>` of capacity 2, let a producer thread put the numbers 1 through 5, then −1 as an end marker. Let `main` take items and add them up until −1 arrives. What is the output?

The producer runs `for (int i = 1; i <= 5; i++) q.put(i);` and `q.put(-1);`. In a loop, `main` does `int v = q.take();`, leaves the loop if `v < 0`, and otherwise does `total += v`. Because the queue is FIFO, −1 always arrives last and all five numbers are added: `Total: 15`.

A capacity of 2 only makes the producer wait more often; it does not change the result. The program was run five times and always printed `Total: 15`.

10.10 Synchronizer objects

10.10.1 Four ready-made tools

The `java.util.concurrent` package offers ready-made classes for common forms of coordination between threads. Each is a solution to a particular scenario.

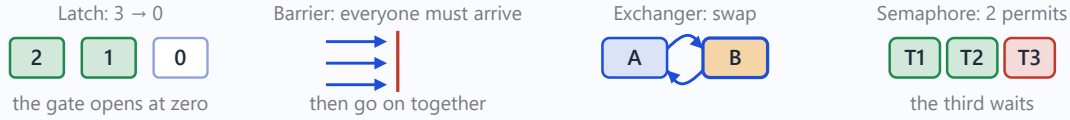


Figure 10.23: `CountDownLatch`, `CyclicBarrier`, `Exchanger`, and `Semaphore`.

`CountDownLatch`
: "Wait until N events have happened, then go on." Waiting threads stay put until the counter reaches zero. It is single-use.

`CyclicBarrier`
: "Let all of us reach the meeting point, then continue together." It can be reused.

`Exchanger`
: Two threads meet at a point and swap data.

`Semaphore`
: N permits: at most N threads can enter the resource at the same time.

10.10.2 `CountDownLatch`

A countdown latch coordinates threads that wait for a certain number of events. An initial count is given to the constructor. Each `countDown()` decreases the counter by one. A thread that calls `await()` waits until the counter reaches 0.

```
public static void main(String[] args)
        throws Exception {
    int count = 3;
    CountDownLatch latch = new CountDownLatch(count);
    for (int i = 0; i < count; i++) {
        int no = i + 1;   // copy for the lambda
        new Thread(() -> {
            System.out.println(no + ": event happened");
            // count down when the event happens
            latch.countDown();
        }).start();
    }
    // wait for the threads to finish
    latch.await();
    System.out.println("All events done!");
}
```

The line `int no = i + 1;` in the loop is important. A lambda can access only local variables that are `final` or effectively `final` (assigned once and never changed). The loop variable `i` changes in every iteration and cannot be used directly; `javac` reports this error: `local variables referenced from a lambda expression must be final or effectively final`. `no`, which is created anew in every iteration, never changes.

Step	Line	Variables / state	New output
1	4: `CountDownLatch latch = new CountDownLatch(count);`	counter=3	—
2	11: `}).start();`	counter=3	—
3	14: `latch.await();`	counter=3	—
4	8: `System.out.println(no + ": event happened");` 10: `latch.countDown();`	counter=2	`1: event happened`
5	8: `System.out.println(no + ": event happened");` 10: `latch.countDown();`	counter=1	`3: event happened`
6	8: `System.out.println(no + ": event happened");` 10: `latch.countDown();`	counter=0	`2: event happened`
7	15: `System.out.println("All events done!");`	counter=0	`All events done!`

Table 10.17: One run of the `CountDownLatch` program (the order in the slides).

Three threads are started, and `main` waits in `await()`. Each thread prints its line and decreases the counter by one. In the run in the slides, the order is 1, 3, 2. When the last `countDown()` brings the counter to 0, the latch opens and `main` wakes up.

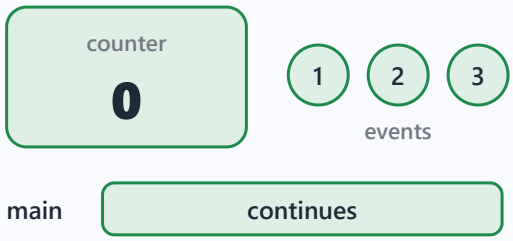


Figure 10.24: Step 6. The last event has happened and the counter is 0; the latch has opened and `main` is waking up.

The line `All events done!` is always last; the order of the event lines varies. The program was run 60 times; the orders seen:

```
13 times:  1 · 3 · 2
13 times:  1 · 2 · 3
12 times:  2 · 3 · 1
 9 times:  3 · 2 · 1
 7 times:  3 · 1 · 2
 6 times:  2 · 1 · 3
```

The latch is single-use. The counter never goes up and cannot be reset. A `countDown()` call when the counter is at 0 has no effect; the counter does not go negative. On a latch that has already opened, `await()` returns without waiting.

10.10.3 `CyclicBarrier`

A cyclic barrier makes a group of threads wait at a meeting point. The constructor receives the number of threads to wait for, called the number of **parties**, and an optional barrier action.

```
public static void main(String[] args) {
    int parties = 3;
    Runnable action = () -> {
        System.out.println("Party complete!");
    };
    CyclicBarrier barrier =
            new CyclicBarrier(parties, action);
    for (int i = 0; i < parties; i++) {
        int no = i + 1;
        new Thread(() -> {
            try {
                System.out.println(no + " waiting");
                // wait for the others
                barrier.await();
                System.out.println(no + " continuing");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

```
1 waiting
2 waiting
3 waiting
Party complete!
3 continuing
1 continuing
2 continuing
```

Each thread waits when it calls `await()`. When the last member of the party also calls `await()`, the barrier action runs once, and then everyone continues. The thread that completes the party runs the action; this was tested separately. This output is one run as well. The line `Party complete!` always comes after the three `waiting` lines and before the three `continuing` lines. The order inside each group varies: in 60 runs, 8 different outputs were seen.

The barrier is "cyclic" in that after the party is full and everyone has passed, the same barrier can be used again. A typical example is players in a game waiting for one another at the end of every round. If one of the waiting threads gets an exception, times out, or is interrupted, the barrier is broken; the other waiting threads receive `BrokenBarrierException`.

10.10.4 `Exchanger`

`Exchanger` is used for exchanging data between two threads. It is more a communication object than a lock.

```
static void swap(Exchanger<String> ex, String data, int ms) {
    try {
        System.out.println("sending: " + data);
        Thread.sleep(ms);
        String got = ex.exchange(data);   // wait for the other
        System.out.println(data + " owner got: " + got);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
public static void main(String[] args) {
    Exchanger<String> exchanger = new Exchanger<>();
    // The first thread sends "Data 1" after 2 s
    new Thread(() -> swap(exchanger, "Data 1", 2000)).start();
    // The second thread sends "Data 2" after 1 s
    new Thread(() -> swap(exchanger, "Data 2", 1000)).start();
```

```
sending: Data 2
sending: Data 1
Data 1 owner got: Data 2
Data 2 owner got: Data 1
```

`exchange(data)` waits until the other side also calls `exchange`. The owner of `Data 2` arrives at `exchange` after one second and waits one more second. When the owner of `Data 1` arrives after two seconds, they meet and each receives the other's data. The order of the first two lines depends on the order in which the two threads start; the last two lines also race at the moment of the meeting. In ten runs, 4 different outputs were seen.

10.10.5 `Semaphore`

Definition 10.16

A **semaphore** is a counter that allows at most N threads to access a resource at the same time. `acquire()` takes a permit; if there is none, it waits. `release()` gives a permit back.

```
public static void main(String[] args) {
    int permits = 3;
    Semaphore semaphore = new Semaphore(permits);
    for (int i = 0; i < 5; i++) {
        int no = i + 1;
        new Thread(() -> {
            try {
                semaphore.acquire();   // take a permit
                System.out.println(no + " in");
                Thread.sleep(2000);
                System.out.println(no + " out");
                semaphore.release();   // give it back
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

Step	Line	Variables / state	New output
1	3: `Semaphore semaphore = new Semaphore(permits);`	permits=3	—
2	16: `}).start();`	permits=3	—

3	8: `semaphore.acquire(); // take a permit`
	9: `System.out.println(no + " in");`
		permits=2	`1 in`

4	8: `semaphore.acquire(); // take a permit`
	9: `System.out.println(no + " in");`
		permits=1	`3 in`

5	8: `semaphore.acquire(); // take a permit`
	9: `System.out.println(no + " in");`
		permits=0	`2 in`

6	8: `semaphore.acquire(); // take a permit`	permits=0	—
7	11: `System.out.println(no + " out");` 12: `semaphore.release(); // give it back`	permits=1	`2 out`
8	11: `System.out.println(no + " out");` 12: `semaphore.release(); // give it back`	permits=2	`1 out`

9	8: `semaphore.acquire(); // take a permit`
	9: `System.out.println(no + " in");`
		permits=0	`4 in ⏎ 5 in`

10	11: `System.out.println(no + " out");` 12: `semaphore.release(); // give it back`	permits=3	`3 out ⏎ 4 out ⏎ 5 out`

Table 10.18: A semaphore with three permits and five threads (the run in the slides).

Five threads are started. The first three each take a permit and enter; in the run in the slides these are 1, 3, and 2. Because no permits are left, 4 and 5 wait in `acquire()`.

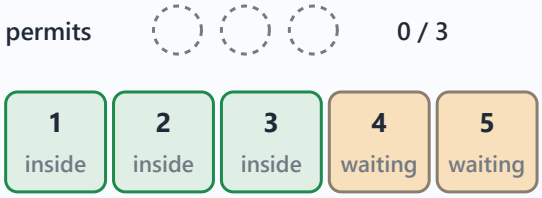


Figure 10.25: Step 6. All three permits are in use; 4 and 5 are waiting in `acquire()`.

Two seconds later, those inside leave and release their permits. The waiting threads 4 and 5 take permits and enter, and two seconds later they leave too. In the end, all three permits are back. Never are more than three threads inside at the same time. Which threads enter first and the order of the lines vary: in 12 runs, 12 different outputs were seen. In every run, the first three lines were three `in` lines, and the fourth entry came only after an exit. The first to enter were not always 1, 2, and 3; in two runs, 4 was also among the first three. Because the output lines are printed outside any lock, an `in` line sometimes appeared even before the `out` line of the thread that had released the permit.

`new Semaphore(n, true)` creates a fair semaphore: permits are handed out in order of waiting. Two properties of the semaphore are surprising. A semaphore does not keep track of who owns the permits; a thread that never acquired a permit can also call `release()`. And the initial count is not an upper limit: every extra `release()` increases the number of permits.

Example 10.9

In a single thread, these run: `Semaphore s = new Semaphore(2);`, `s.acquire();`, `x1 = s.tryAcquire();`, `x2 = s.tryAcquire();`, then `s.release();` three times. Then `CountDownLatch m = new CountDownLatch(2);` and `m.countDown();` three times. What are `x1`, `x2`, `s.availablePermits()`, `m.getCount()`, and `m.await(10, TimeUnit.MILLISECONDS)`?

The semaphore starts with 2 permits. `acquire()` takes one: 1 left. `tryAcquire()` takes the remaining permit and returns `true`: 0. The second `tryAcquire()` finds no permit and returns `false` without waiting; `acquire()` would have waited forever. Three `release()` calls raise the permits to 3, above the initial 2. The latch drops to 0 with two `countDown()` calls; the third has no effect. On an open latch, `await` returns `true` without waiting. The program printed `true false 3` and `0 true`.

10.11 Other tools

10.11.1 Atomic classes

Using a lock to protect a single counter is like throwing a ball to catch a fly. The `java.util.concurrent.atomic` package offers lock-free, thread-safe counters.

> **Definition 10.17**
>
> An **atomic operation** is an operation that completes in a single step, without being split; no other thread can get into the middle of it. Classes such as `AtomicInteger`, `AtomicLong`, and `AtomicBoolean` offer atomic operations on their values.

```
public class AtomicCounter {
    public static void main(String[] args)
            throws InterruptedException {
        AtomicInteger counter = new AtomicInteger(0);
        Runnable r = () -> {
            for (int i = 0; i < 100_000; i++) {
                counter.incrementAndGet();   // atomic ++counter
            }
        };
        Thread a = new Thread(r), b = new Thread(r);
        a.start(); b.start();
        a.join(); b.join();
        System.out.println("Result: " + counter.get());
        boolean b1 = counter.compareAndSet(200000, 0);
        System.out.println(b1 + " " + counter.get());
        boolean b2 = counter.compareAndSet(5, 10);
        System.out.println(b2 + " " + counter.get());
    }
}
```

```
Result: 200000
true 0
false 0
```

`incrementAndGet()` is the atomic counterpart of `++x`: it increases the value by one and returns the new value. The program was run 10 times and found `Result: 200000` every time. Then come two `compareAndSet` calls. Because the value is 200000, the first one succeeds, writes 0, and returns `true`. The second expects 5, but the value is 0; it writes nothing and returns `false`.

Definition 10.18

Compare-and-set (CAS): `compareAndSet(expected, new)` writes `new` and returns `true` if the value is still `expected`. If the value is different, it leaves it alone and returns `false`. The comparison and the write are a single atomic operation; the processor does this with one instruction.

Roughly speaking, an atomic object is the combination of a `volatile` field and an atomic `compareAndSet` operation. `volatile` provides visibility, and CAS provides indivisibility.

10.11.2 Code trace: `compareAndSet`

Without a lock, how can `incrementAndGet` be safe? The following method writes its logic with CAS.

```
AtomicInteger counter = new AtomicInteger(5);

int inc() {                    // how incrementAndGet works
    while (true) {
        int old = counter.get();
        int next = old + 1;
        if (counter.compareAndSet(old, next))
            return next;
        // failed: someone got in first; retry
    }
}
```

Step	Line	Call stack	Variables / state
1	1: `AtomicInteger counter = new AtomicInteger(5);`	—	counter=5
2	5: `int old = counter.get();`	T1: inc()	T1.old=5
3	5: `int old = counter.get();`	T1: inc() → T2: inc()	T1.old=5, T2.old=5
4	6: `int next = old + 1;`	T1: inc() → T2: inc()	T1.next=6, T2.next=6
5	7: `if (counter.compareAndSet(old, next))` 8: `return next;`	T1: inc()	counter=6, T1 → 6
6	7: `if (counter.compareAndSet(old, next))`	T2: inc()	counter=6, T2 CAS(5, 6)=false
7	5: `int old = counter.get();`	T2: inc()	T2.old=6
8	6: `int next = old + 1;`	T2: inc()	T2.next=7
9	7: `if (counter.compareAndSet(old, next))` 8: `return next;`	T2: inc()	counter=7

Table 10.19: Two threads calling `inc()` at the same time.

In steps 2 and 3, T1 and T2 read the current value; both see 5. Because there is no lock, both carry on. In step 4, both compute 6 locally. With an ordinary `counter++`, both would write 6 and one increment would be lost.

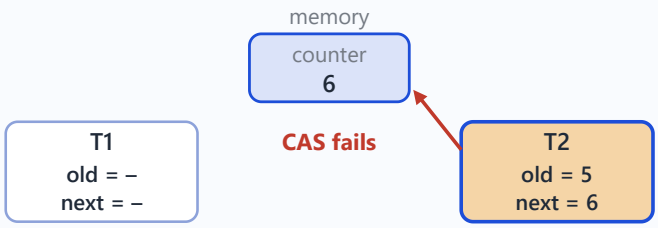


Figure 10.26: Step 6. The CAS of T1 succeeded and the value is 6. The call `compareAndSet(5, 6)` of T2 fails: expected 5, value 6.

In step 5, T1 calls `compareAndSet(5, 6)`. Because the value is still 5, 6 is written and `true` is returned. In step 6, T2 also calls `compareAndSet(5, 6)`. The value is now 6, not the expected 5; nothing is written and `false` is returned. T2 does not lose its increment; it goes back around the loop: in step 7 it rereads the current value, 6, and computes 7. In step 9 the CAS succeeds this time: `counter = 7`. Both increments are counted. Nobody waited for a lock; only T2 went around the loop once more.

When this method was called 100,000 times by each of two threads, the counter reached 5 + 200,000 = 200,005; the result was the same in all five runs. The advantage of the lock-free method is that threads never sleep. When contention is low, it is faster than a lock; under very heavy contention, threads spin for nothing.

10.11.3 Interruption

The second cooperative way to stop a thread is interruption. `t.interrupt()` sends an interrupt request to `t`; it does not stop `t` by force.

Definition 10.19

An **interrupt** is a request sent to a thread to "drop what you are doing". `interrupt()` sets the target's interrupt flag. If the target is waiting in `sleep`, `wait`, or `join`, the wait is cut short immediately and `InterruptedException` is thrown; at that moment the flag is cleared.

There are two ways to read the flag. `Thread.interrupted()` is a `static` method; it reads and clears the flag of the running thread. `isInterrupted()` is an instance method and only reads the flag.

```
public class InterruptDemo {
    public static void main(String[] args)
            throws InterruptedException {
        Thread worker = new Thread(() -> {
            try {
                while (!Thread.interrupted()) {
                    // the thread's work
                    System.out.println("working...");
                    Thread.sleep(400);
                }
            } catch (InterruptedException e) {
                System.out.println("Thread cancelled!");
            }
        });
        worker.start();
        Thread.sleep(1000);
        worker.interrupt();   // interrupt it in sleep()
        worker.join();
        System.out.println("main: done");
    }
}
```

```
working...
working...
working...
Thread cancelled!
main: done
```

The worker stays in the loop as long as the flag is not set. In each iteration it prints a line and sleeps for 400 ms. After one second, `main` calls `interrupt()`.

Step	Line	Variables / state	New output
1	1: `Thread worker = new Thread(() -> {` 11: `worker.start();`	worker=RUNNABLE	—
2	3: `while (!Thread.interrupted()) {` 4: `System.out.println("working...");` 5: `Thread.sleep(400);`	worker=TIMED_WAITING	`working...`
3	12: `Thread.sleep(1000);`	main=TIMED_WAITING	—
4	3: `while (!Thread.interrupted()) {` 4: `System.out.println("working...");` 5: `Thread.sleep(400);`	time=≈400 ms	`working...`
5	3: `while (!Thread.interrupted()) {` 4: `System.out.println("working...");` 5: `Thread.sleep(400);`	time=≈800 ms	`working...`
6	13: `worker.interrupt(); // worker is in sleep()`	time=≈1000 ms	—

Step	Line	Variables / state	New output
7	7: `} catch (InterruptedException e) {` 8: `System.out.println("cancelled!");`	e=InterruptedException	`cancelled!`
8	14: `worker.join();`	worker=TERMINATED	—
9	15: `System.out.println("main: done");`	—	`main: done`

Table 10.20: Trace of the interrupt program.

The worker prints at 0, about 400, and about 800 ms. Its third sleep would end at 1200 ms. `main` wakes up at 1000 ms and calls `interrupt()`. In step 7 the worker is inside `sleep`; the sleep is cut short immediately, `InterruptedException` is thrown, and the flag becomes `false` again.



Figure 10.27: Step 7. `interrupt()` caught the worker inside `sleep`; the sleep was cut short and the `catch` ran.

The `catch` block prints its message, and `run()` ends. `main` waits for the worker to finish with `join()` and prints its last line. The program was run ten times and always printed three `working...` lines. An interrupt is a request: the thread handles the flag or the exception itself and exits.

Example 10.10

While a thread is in `Thread.sleep(5000)`, `main` sends it `interrupt()`. Inside the `catch`, `Thread.currentThread().isInterrupted()` is printed. Then `main` sets its own flag with `Thread.currentThread().interrupt()` and prints `Thread.interrupted()` twice. What is the output?

`sleep` clears the flag when it throws `InterruptedException`; inside the `catch`, `isInterrupted()` is `false` and `interrupted` `false` is printed. The first call of `Thread.interrupted()` in `main` reads the set flag and clears it: `true`. The second reads the cleared flag: `false`. The program printed `interrupted` `false` and `true` `false` in all three runs.

Common mistake

Swallowing `InterruptedException` with an empty `catch`. When the exception is thrown, the flag has been cleared. If the `catch` does nothing, the interrupt request is lost and the thread keeps running. The right thing is to leave the loop or to pass the request upward by setting the flag again inside the `catch` with `Thread.currentThread().interrupt()`.

10.11.4 Suspension

In old versions of Java, `suspend()` froze a thread and `resume()` continued it. Because `suspend()` froze the thread together with the locks it held, it was very prone to deadlock; on JDK 21, both methods throw `UnsupportedOperationException`. The same job is done safely with `wait()` and `notifyAll()`.

```
class Task implements Runnable {
    private boolean paused = false;
    public synchronized void pause() {
        paused = true;
    }
    public synchronized void resume() {
        paused = false;
        notifyAll();   // wake the waiter
    }
    public void run() {
        try {
            for (int i = 1; i <= 5; i++) {
                synchronized (this) {
                    while (paused) wait();
                }
                System.out.println("step " + i);
                Thread.sleep(200);
            }
        } catch (InterruptedException e) { }
    }
}
```

At the start of each step, the task checks the `paused` flag under the lock. If the flag is raised, it waits with `wait()`. `resume()` lowers the flag and wakes the waiting thread with `notifyAll()`. The flag is protected by the same monitor in both methods; it does not also need to be `volatile`.

```
Task g = new Task();
Thread t = new Thread(g);
t.start();
Thread.sleep(500);
g.pause();
System.out.println("main: paused");
Thread.sleep(1000);
System.out.println("main: resume");
g.resume();
t.join();
```

```
step 1
step 2
step 3
main: paused
main: resume
step 4
step 5
```

The task prints a step at 0, 200, and 400 ms. `main` pauses it at 500 ms. When the task wakes at 600 ms, it sees the flag and waits. For one second, no step is printed. When `main` resumes it, the remaining two steps are printed. The program was run ten times and always printed these seven lines. The pause takes effect between steps, not in the middle of one; so no object is left half finished.

10.11.5 `ThreadLocal`

The surest remedy for a race is not to share. `ThreadLocal` gives each thread its own copy of the same variable.

```
static final ThreadLocal<String> value =
        new ThreadLocal<>();
public static void main(String[] args)
        throws InterruptedException {
    Thread thread1 = new Thread(() -> {
        // first thread
        value.set("Value 1");
        System.out.println("thread1: " + value.get());
    });
    Thread thread2 = new Thread(() -> {
        // second thread
        value.set("Value 2");
        System.out.println("thread2: " + value.get());
    });
    thread1.start();
    thread2.start();
    thread1.join();
    thread2.join();
    System.out.println("main: " + value.get());
```

```
thread1: Value 1
thread2: Value 2
main: null
```

Both threads write to the same `value` object, but each writes to and reads from its own copy. The copies are independent of each other. Because `main` never called `set`, its own copy is empty and it sees `null`. The order of the first two lines varies: in 40 runs, `thread1` printed first 20 times and `thread2` 20 times. The line `main: null` was always last.

To give each thread a separate initial value, write `ThreadLocal.withInitial(() -> 0)` (Java 8+) or override the `initialValue()` method. A typical use is to give each thread its own copy of an object that is not thread-safe, such as a date formatter.

Example 10.11

`ThreadLocal<Integer> t = ThreadLocal.withInitial(() -> 0);`, then `t.set(5)` in `main`. A new thread `x` does `t.set(t.get() + 1)` and prints `"x: " + t.get()`; `main` waits for `x` and prints `"main: " + t.get()`. What is the output?

The 5 in `main` does not pass to `x`. On its first `get()`, `x` starts with the 0 from `withInitial` and makes it 1: `x: 1`. What `x` wrote does not affect the copy of `main` either: `main: 5`.

10.11.6 Thread pools

Starting a new thread for every task creates two problems. Creating a thread is not cheap: a stack is allocated and an operating-system record is made. More importantly, the number grows without control: if a thousand requests arrive, a thousand threads are started.

Definition 10.20

A **thread pool** is a structure that creates the threads to be used in advance and distributes tasks to them. Tasks are submitted to the pool (`submit`); an idle worker takes a task and, when finished, moves on to the next one. If there is no idle worker, the task waits in the queue.

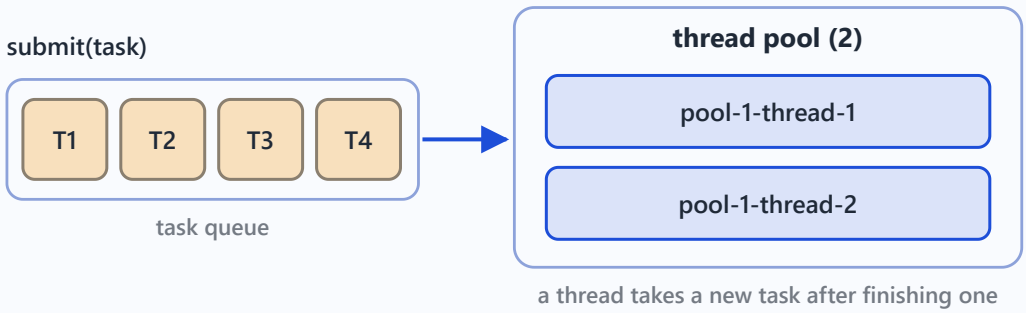


Figure 10.28: Tasks are added to a queue; the two threads in the pool take tasks in turn.

A pool reuses threads and limits their number. In Java, pools are managed through the `ExecutorService` interface. `Executors.newFixedThreadPool(n)` gives a ready-made pool with a fixed `n` workers. When detailed configuration is needed, the `ThreadPoolExecutor` class, which creates and manages the pool, is used directly.

```
public class PoolDemo {
    static void task(int no) {
        String name = Thread.currentThread().getName();
        System.out.println("Task " + no + " -> " + name);
    }
    public static void main(String[] args) throws Exception {
        ExecutorService pool = Executors.newFixedThreadPool(2);
        for (int i = 1; i <= 4; i++) {
            int no = i;
            pool.submit(() -> task(no));
        }
        // Callable: a task that returns a value
        Future<Integer> f = pool.submit(() -> 6 * 7);
        System.out.println("Callable result: " + f.get());
        pool.shutdown();   // accept no new tasks
        pool.awaitTermination(5, TimeUnit.SECONDS);
        System.out.println("Pool shut down");
    }
}
```

The first run in the slides is:

```
Task 1 -> pool-1-thread-1
Task 2 -> pool-1-thread-2
Task 3 -> pool-1-thread-2
Task 4 -> pool-1-thread-2
Callable result: 42
Pool shut down
```

The four tasks are distributed over the two workers. Which worker gets which task and the order of the lines vary; in a second run, the slides give the last two task lines as `Task 3 -> pool-1-thread-1` and `Task 4 -> pool-1-thread-2`. For these notes, 6 different outputs were seen in 40 runs. Some things do not change: the worker names are always `pool-1-thread-1` and `pool-1-thread-2`; the pool never starts a third thread. The last two lines are always the same, too.

`Runnable` does not return a value. For tasks that return a value there is the `Callable` interface; the lambda `() -> 6 * 7` is a `Callable<Integer>`. `submit` returns a `Future` immediately. A `Future` represents a result that will be ready in the future. `get()` waits until the result is ready.

`shutdown()` asks the pool to stop accepting new tasks; the tasks in the queue are still completed. `awaitTermination` waits for all tasks to finish, for at most the given time. If `shutdown()` is forgotten, the program stays open, because the pool's workers are not daemon threads.

Java 21 introduced a new option: `Executors.newVirtualThreadPerTaskExecutor()` gives each task its own **virtual thread**. Virtual threads are managed by the JVM, not by the operating system; millions of them can be started. They are suited to large numbers of short, wait-heavy tasks. In one

experiment, `isVirtual()` and `isDaemon()` both returned `true` inside such a task.

10.11.7 Code trace: pool and `Future`

To see how the pool works over time, the tasks imitate long jobs by sleeping. `job(ms, d)` sleeps for `ms` milliseconds and returns `d`.

```
static int job(int ms, int d) throws Exception {
    Thread.sleep(ms);   // stands in for slow work
    return d;
}
// inside main:
ExecutorService pool = Executors.newFixedThreadPool(2);
Future<Integer> f1 = pool.submit(() -> job(300, 10));
Future<Integer> f2 = pool.submit(() -> job(100, 20));
Future<Integer> f3 = pool.submit(() -> job(100, 30));
System.out.println(f1.isDone() + " " + f3.isDone());
int sum = f1.get() + f2.get() + f3.get();
System.out.println("sum = " + sum);
System.out.println(f2.isDone() + " " + f3.isDone());
pool.shutdown();
```

Step	Line	Variables / state	New output
1	6: `ExecutorService pool = Executors.newFixedThreadPool(2);`	pool=2 workers	—
2	7: `Future<Integer> f1 = pool.submit(() -> job(300, 10));`	f1=Future (pending)	—
3	8: `Future<Integer> f2 = pool.submit(() -> job(100, 20));`	f2=Future	—
4	9: `Future<Integer> f3 = pool.submit(() -> job(100, 30));`	f3=Future	—
5	10: `System.out.println(f1.isDone() + " " + f3.isDone());`	f1.isDone()=false	`false false`
6	11: `int sum = f1.get() + f2.get() + f3.get();`	time=≈100 ms	—
7	11: `int sum = f1.get() + f2.get() + f3.get();`	time=≈200 ms	—
8	11: `int sum = f1.get() + f2.get() + f3.get();`	sum=10+20+30=60	—
9	12: `System.out.println("sum = " + sum);` 13: `System.out.println(f2.isDone() + " " + f3.isDone());`	sum=60	`sum = 60` ⏎ `true true`
10	14: `pool.shutdown();`	pool=shutting down	—

Table 10.21: Trace of three tasks in a pool with two workers (times approximate).

The pool has two workers. `f1` starts on worker 1 and `f2` on worker 2. Because both workers are busy, `f3` waits in the queue; no new thread is started. In step 5, no task has finished; `isDone()` returns `false` without waiting. In step 6, `f1.get()` stops `main` until the result is ready. Meanwhile, at about 100 ms, `f2` finishes and worker 2 takes `f3` from the queue.

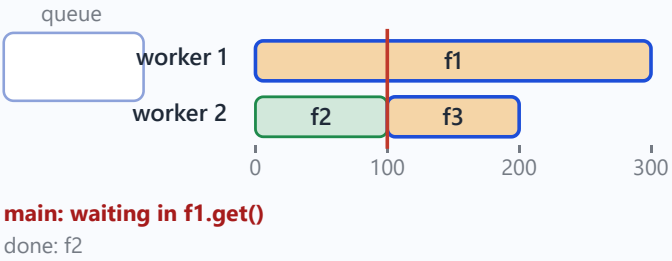


Figure 10.29: Step 6. `main` is waiting in `f1.get()`; `f2` is done and worker 2 has taken `f3` from the queue.

At about 200 ms, `f3` finishes too; `main` is still waiting for `f1`. At 300 ms, `f1` is ready. Because `f2` and `f3` have already finished, their `get()` calls return without waiting: total 10 + 20 + 30 = 60. If the tasks had run one after another in a single thread, it would have taken 300 + 100 + 100 = 500 ms; with the pool the total time is about 300 ms. The program was run ten times; the output was always `false false`, `sum = 60`, `true true`, and the measured time was between 300 and 350 ms every time. Which worker gets which task may vary from run to run.

10.11.8 `CompletableFuture` and concurrent collections

```
CompletableFuture<Integer> cf = CompletableFuture
    .supplyAsync(() -> 6)
    .thenApply(x -> x * 7)              // 42
    .thenCombine(CompletableFuture
        .supplyAsync(() -> 8), Integer::sum);
System.out.println(cf.get());           // 50

Map<String, Integer> freq =
    new ConcurrentHashMap<>();
String[] w = {"apple", "pear", "apple"};
Arrays.stream(w).parallel()
    .forEach(s -> freq.merge(s, 1, Integer::sum));
System.out.println(freq);  // {apple=2, pear=1}
```

This code shows two different tools, and in all ten runs it printed `50` and `{apple=2, pear=1}`.

`CompletableFuture`

Instead of waiting with `get()`, it chains operations; each step runs in the background when the previous one finishes. In the code, 6 is produced, multiplied by 7 (42), and added to the 8 produced by a separate task: 50.

`ConcurrentHashMap`
: A thread-safe map. The methods `merge` and `compute` are atomic. A parallel stream processes the elements of the array on different threads; `merge` makes sure that no count is lost.

`CopyOnWriteArrayList`
: Copies the internal array on every write. It suits lists that are read a lot and written rarely.

`ForkJoinPool`
: For divide-and-conquer tasks (`RecursiveTask`). An idle worker steals work from another worker's queue (work stealing). Parallel streams (`parallelStream()`) use this pool.

A parallel stream is not always fast. On small data, the cost of coordinating threads exceeds the gain. A parallel stream that writes to a shared mutable variable, on the other hand, corrupts the result. The decision to parallelize is made by measurement.

10.11.9 Thread safety

> **Definition 10.21**
>
> A class is **thread-safe** if it behaves correctly when used from several threads at the same time.

Thread safety is reached in four ways:

Immutable objects
: Objects that do not change after they are created (`String` , classes whose fields are all `final` , Java 16+ `record`) can be shared without synchronization. Data that does not change does not race.

Not sharing
: Local variables live on each thread's own stack; `ThreadLocal` also gives each thread a copy.

Synchronization
: Every access to shared mutable data is protected by the same lock: `synchronized` or `Lock` .

Ready-made structures
: `AtomicInteger` , `ConcurrentHashMap` , `CopyOnWriteArrayList` , `BlockingQueue` .

The ordinary classes of the collections framework, `ArrayList` and `HashMap` , are not thread-safe. When two threads each added 10,000 elements to the same `ArrayList` , the size was not 20,000 in any of the ten runs made for these notes; the values found ranged from 10,004 to 14,121. In four runs, one of the threads died with `ArrayIndexOutOfBoundsException` while the internal array was being grown. These classes are either protected with a lock or replaced by their concurrent counterparts.

10.12 Worked problems

Example 10.12 (Problem 1)

What is the output? Is it the same in every run?

```
Thread t = new Thread(() -> System.out.print("B"));
System.out.print("A");
t.start();
t.join();
System.out.print("C");
```

`ABC`, always. `A` is printed before the thread is started. `B` is printed in the new thread; `join()` does not let `main` move on to `C` until `B` has been printed and the thread has finished. In all ten runs, the output was `ABC`.

Example 10.13 (Problem 2)

What is the output?

```
Thread t = new Thread(() -> System.out.print("X"));
System.out.print(t.getState() + " ");
t.run();
System.out.print(" " + t.getState());
```

`NEW X NEW`. `run()` is an ordinary call: `main` prints the `X`, the thread never starts, and it stays `NEW`.

Example 10.14 (Problem 3)

Which of the following are possible outputs of the "two threads" example? a) `A1 A2 B1 B2 main`; b) `B1 A1 A2 B2 main`; c) `A1 main B1 A2 B2`; d) `A2 A1 B1 B2 main`. (Abbreviation: `A1` is the line `A: step 1`.)

a and b. c is impossible, because `main` prints after both `join()` calls. d is impossible, because the steps of the same thread keep their order. a was the most frequent order in the measurements made for these notes. The mirror image of b, `A1 B1 B2 A2`, was also seen twice in the 6,000 trials inside a single JVM.

Example 10.15 (Problem 4)

What is the state of the thread in each case? a) inside `sleep(500)`; b) trying to enter a `synchronized` block that is held; c) after `wait()`; d) when `run()` has ended.

a) `TIMED_WAITING`; b) `BLOCKED`; c) `WAITING`; d) `TERMINATED`. In c, the thread is in the wait set. Once `notify` arrives and it is trying to reacquire the lock, it becomes `BLOCKED`.

Example 10.16 (Problem 5)

Is the following class thread-safe?

```
class Account {
    private int balance = 0;
    void deposit(int m) { balance += m; }
    int getBalance() { return balance; }
}
```

No: `balance += m` is a read–add–write. When two threads each called `deposit(1)` 100,000 times, the balance was not 200,000 in any of the ten runs; the results ranged from 126,397 to 171,138. The fix: make the methods `synchronized`, or make the field an `AtomicInteger` and use `addAndGet(m)`.

Example 10.17 (Problem 6)

Two threads run these blocks. What can happen?

```
// T1                    // T2
synchronized (a) {       synchronized (b) {
  synchronized (b) { }     synchronized (a) { }
}                        }
```

Deadlock: if T1 takes `a` and T2 takes `b`, each waits forever for the other. Because there is no sleep, the deadlock does not happen in every run; that is what makes it dangerous. When the blocks were run thousands of times each in a loop, the deadlock was seen in all three trials. The fix: both threads take `a` first and then `b`.

Example 10.18 (Exercise: two half sums)

The sum of the numbers 1..1000 will be computed with two threads: one adds the first half, the other the second half; `main` waits for both and prints the total.

```
Thread left = new Thread(() -> {
    for (int i = 0; i < 500; i++) part[0] += arr[i];
});
Thread right = new Thread(() -> {
    for (int i = 500; i < 1000; i++) part[1] += arr[i];
});
left.start(); right.start();
left.join(); right.join();
```

Each thread writes to its own slot; because the two threads never write to the same slot, there is no race. Reading after `join()` is safe. With `arr[i] = i + 1`, the parts are 125,250 and 375,250; the total is 500,500 = 1000 · 1001 / 2.

Example 10.19 (Exercise: counting even numbers)

The even numbers between 1 and 1000 will be counted in a shared counter by four threads, each taking 250 numbers.

The counter is an `AtomicInteger evens`. Each thread runs `for (int i = start; i < start + 250; i++) if (i % 2 == 0) evens.incrementAndGet();`; the `start` values are 1, 251, 501, and 751. After all the `join()` calls, the output is `Even numbers: 500`. If the counter were an ordinary `int`, the result could come out smaller than 500 because of the race.

Example 10.20 (Exercise: sum of squares)

The squares of the numbers 1..5 will be computed with `Callable` tasks in a pool of 3 threads and collected from the `Future`s.

```
ExecutorService pool = Executors.newFixedThreadPool(3);
List<Future<Long>> results = new ArrayList<>();
for (int i = 1; i <= 5; i++) {
    int n = i;
    results.add(pool.submit(() -> square(n)));
}
long sum = 0;
for (Future<Long> f : results) sum += f.get();
pool.shutdown();
System.out.println("Sum of squares: " + sum);
```

The helper method is `static long square(int n) { return (long) n * n; }`. The `Future`s stay in the list in the order they were submitted; `get()` waits for the result of each. The output is `Sum of squares: 55` (1 + 4 + 9 + 16 + 25).

Example 10.21 (Exercise: a clock)

Write a thread that prints `tick` every second; after 3.5 seconds, `main` will stop it with `interrupt()`.

The pattern from the interrupt example is used: `while (!Thread.interrupted())` in the loop, and in the body print `tick` and call `sleep(1000)`; after the `catch`, print `clock stopped`. `main` runs `sleep(3500)`, `interrupt()`, and `join()`. The thread prints at seconds 0, 1, 2, and 3; at second 3.5 its sleep is cut short. The output is four `tick` lines and `clock stopped`; it was the same in all five runs.

10.13 Common mistakes

The following two tables collect the common mistakes from the slides. The compiler messages were verified with JDK 21's `javac`, and the run-time symptoms by running the programs.

Mistake	Symptom	Correct form
Calling `t.run()`	No new thread; the code runs in order in `main`	`t.start()`
Calling `start()` twice	`IllegalThreadStateException`	A new `Thread` for each run
`Thread.sleep(100);` on its own	`error: unreported exception InterruptedException; must be caught or declared to be thrown`	`try`-`catch` or `throws InterruptedException`
The loop variable `i` inside a lambda	`error: local variables referenced from a lambda expression must be final or effectively final`	A copy: `int no = i;`
Catching an exception that is never thrown: `try { while (!Thread.interrupted()) { } } catch (InterruptedException e)`	`error: exception InterruptedException is never thrown in body of corresponding try statement`	The body must contain `sleep` / `wait`, or the `catch` must be removed
`currentThread()` in a class that does not extend `Thread`	`error: cannot find symbol`	`Thread.currentThread()`
Forgetting `join()`	`main` reads before the result is ready (e.g., 0)	`join()` before reading the result

Table 10.22: Mistakes in creating and managing threads.

Mistake	Symptom	Correct form
Calling `wait()` / `notify()` outside `synchronized`	`IllegalMonitorStateException: current thread is not owner`	Inside `synchronized` on the same object
Waiting for the condition with `if`	The condition may be invalid again on wakeup; a queue that overflows or is read while empty	`while (condition) wait();`

Mistake	Symptom	Correct form
Writing `unlock()` outside `finally`	On an exception the lock is never released; everyone waits forever	`lock(); try { … } finally { unlock(); }`
Unlocking a lock that was not acquired	`IllegalMonitorStateException`	One `unlock()` for every `lock()`
`counter++` on a `volatile` counter	Increments are lost (e.g., 163299)	`AtomicInteger` or `synchronized`
A `final volatile` field	`error: illegal combination of modifiers: final and volatile`	Use only one of them
Using `stop()` or `suspend()`	Java 20+: `UnsupportedOperationException`	A flag, `interrupt()`, or `wait` / `notify`
Swallowing `InterruptedException` with an empty `catch`	The interrupt request is lost and the thread does not stop	Leave the loop, or call `Thread.currentThread().interrupt()`
Acquiring locks in different orders	Deadlock: the program freezes	Everyone acquires them in the same order

Table 10.23: Synchronization mistakes.

The compiler catches some of these mistakes; the rest appear only at run time, and often only in some runs. Race and deadlock bugs become visible at moments of high load, for example at peak hours on a server. That is why "I tried it and it worked" is not proof for multithreaded code. For every shared variable, it must be clearly known which lock protects it.

10.14 Summary

- A thread is an independently progressing flow of execution inside a process. The threads of one process share the heap and the method area; each has its own stack, PC register, and local variables.
- Concurrency means that tasks make progress during the same period of time, and it happens even on a single core. Parallelism means that tasks really run at the same instant on different cores.
- The scheduler decides which thread runs when. The output order of multithreaded programs can therefore vary from run to run.
- A thread is written by implementing `Runnable`, by extending `Thread`, or with a lambda. The work goes into `run()`; the thread is started with `start()`. Calling `run()` directly does not start a new thread.
- There are six states: `NEW`, `RUNNABLE`, `BLOCKED`, `WAITING`, `TIMED_WAITING`, `TERMINATED`. A thread that has finished cannot be restarted.
- `join()` waits for a thread to finish and makes what it wrote visible. `sleep` does not release locks.
- Daemon threads do not prevent the JVM from exiting; `setDaemon(true)` is called before `start()`.

- A thread is stopped cooperatively: with a `volatile` flag or with `interrupt()`. `stop()` has not worked since Java 20.
- A race condition arises from divisible operations on shared data. `counter++` consists of read–add–write steps, and a thread that slips in between makes increments get lost.
- `synchronized` provides mutual exclusion by acquiring an object's monitor. An instance method locks `this`, and a `static` method locks `Class.class`. The protection works only if everyone uses the same lock.
- `volatile` provides visibility, not atomicity. Happens-before relations (lock, `volatile`, `start`, `join`) determine when what one thread wrote becomes visible to another.
- `Lock` is an explicit lock: it offers `tryLock`, timed and interruptible waiting, fairness, and several `Condition`s. `unlock()` always goes in `finally`. `ReentrantLock` and `synchronized` are reentrant.
- `ReadWriteLock` lets readers read at the same time and gives the writer exclusive access.
- Deadlock arises when locks are acquired in different orders. A fixed lock order, timed attempts, and short critical sections prevent it.
- To wait for a condition, use `wait` / `notifyAll` or the `await` / `signal` of `Condition`; check the condition with `while`. The ready-made `ArrayBlockingQueue` solves the producer–consumer problem by itself.
- `CountDownLatch` counts events, `CyclicBarrier` makes threads meet, `Exchanger` swaps data, and `Semaphore` limits concurrent entry.
- Atomic classes update safely and without locks using `compareAndSet`. `ThreadLocal` gives each thread its own copy.
- Thread pools reuse threads and limit their number. `Callable` returns a value, `Future.get()` waits for the result, and `shutdown()` must not be forgotten.
- The ways to thread safety: immutable objects, not sharing, synchronization, and ready-made concurrent structures. `ArrayList` and `HashMap` are not thread-safe.

10.15 Key terms

thread process concurrency / parallelism thread states daemon thread

Java Virtual Machine (JVM) functional interface lambda expression race condition synchronization

monitor mutual exclusion lock deadlock producer–consumer problem semaphore

atomic operation thread pool thread safety immutable context switching scheduler

critical section visibility, `volatile` happens-before reentrant lock livelock / starvation wait set

compare-and-set interrupt

10.16 Test Yourself

1. A `static` method defines `int total = 0;`, increments `total++` 1,000 times in a loop, and prints the result. If two threads named A and B run this method, what does each print? Is there a race?

   Answer

   Both print 1000. `total` is a local variable and lives on each thread's own stack; the two copies never see each other. A race happens only when the writes go to a shared field (an object field or a `static` field).

2. A class has `static synchronized void s() { shared++; }` and `synchronized void o() { shared++; }`; `shared` is a `static int`. One thread calls `s()` 100,000 times, and another calls `o()` 100,000 times on the same object. Is the result always 200,000?

   Answer

   No. `s()` locks the class's `.class` object, and `o()` locks `this`. Two different locks do not block each other, and `shared++` can run in both threads at the same time. In five trials, the result was below 200,000 every time (between 180,586 and 198,518). The fix is for both methods to use the same lock.

3. On a `ReentrantLock k`, `main` calls `lock()` twice and `unlock()` once. Then another thread calls `k.tryLock()`. What does it return? What is `k.getHoldCount()` in `main`?

   Answer

   `tryLock()` returns `false`; `getHoldCount()` is 1. The hold count has dropped from 2 to 1, and the lock is still held by `main`. The lock is released only by the second `unlock()`.

4. `countDown()` is called twice on `CountDownLatch m = new CountDownLatch(3);`. What does `m.await(100, TimeUnit.MILLISECONDS)` return? What is `m.getCount()`?

   Answer

   `false` and 1. Because the counter has not reached 0, the latch does not open; the timed `await` waits 100 ms and returns `false` when the time is up. If the untimed `await()` had been called, the thread would have waited until the third event arrived.

5. Three tasks are submitted to a pool created with `Executors.newFixedThreadPool(1)`, each printing its own number and the name of its thread. What are the output order and the thread

names?

Answer

Because the pool has a single worker, the tasks run one after another in queue order: 1, 2, 3. All three run on `pool-1-thread-1`; the pool starts no new thread. A single-worker pool therefore preserves the order of tasks. In five trials, the output was always the same.

6. `AtomicInteger a = new AtomicInteger(10);` is created, then `a.compareAndSet(10, 20)`, `a.compareAndSet(10, 30)`, and `a.getAndIncrement()` are called. What do the three calls return, and what is the final value of `a`?

Answer

`true`, `false`, `20`; the final value is 21. The first CAS finds 10 and writes 20. The second expects 10, but the value is 20, so it does not touch it. `getAndIncrement()` returns the old value (20) first and then increments.

7. `main` first calls `Thread.currentThread().interrupt()`, then runs `Thread.sleep(1000)` inside a `try`-`catch`. Does `sleep` last a second? What is `isInterrupted()` inside the `catch`?

Answer

No. If a thread whose interrupt flag is set enters `sleep`, `InterruptedException` is thrown immediately. In the experiment, the exception came in under 100 ms. When the exception is thrown, the flag is cleared; `isInterrupted()` is `false`.

8. Three threads acquire three locks as follows: T1 takes `a` then `b`; T2 takes `b` then `c`; T3 takes `c` then `a`. Can deadlock occur? If it can, which rule prevents it?

Answer

It can. If all three acquire their first lock, T1 waits for `b`, T2 for `c`, and T3 for `a`. Each lock being waited for is held by the next thread, and the cycle T1 → T2 → T3 → T1 forms. A fixed order is imposed, for example `a` < `b` < `c`, and every thread acquires its locks in that order. T3 then takes `a` first and `c` second, and no cycle can form.

9. In the blocking-queue example, what problem could arise if `notify()` were written instead of

`notifyAll()`?

Answer

Both producers and consumers can be waiting in the same wait set. `notify()` chooses a single thread arbitrarily. If a consumer, after taking an item, wakes another consumer, that consumer finds the queue empty and waits again. The waiting producer never wakes up, and the program may hang. `notifyAll()` wakes everyone; the irrelevant threads find the `while` condition unsatisfied and wait again. If producers and consumers are given separate sets with `Condition`, `signal()` is enough.

10.17 Further reading

- Oracle, *The Java Tutorials*, "Lesson: Concurrency": an official, free treatment of processes and threads, synchronization, liveness problems, guarded blocks, and high-level concurrency objects.
- B. Goetz et al., *Java Concurrency in Practice*: the standard book on thread safety, visibility, locks, and pools; its examples are still valid today.
- J. Bloch, *Effective Java* (3rd ed.), Chapter 11, Items 78–84: explains, with reasons, how to synchronize access to shared mutable data, why to prefer the ready-made utilities over `wait`/`notify`, and how to avoid excessive synchronization.
- J. Gosling et al., *The Java Language Specification, Java SE 21 Edition*, Chapter 17, "Threads and Locks": the precise definition of happens-before and of `volatile`.

10.18 References

1. Oracle, *The Java Tutorials*: Lesson: Concurrency (Processes and Threads, Synchronization, Liveness, Guarded Blocks, High Level Concurrency Objects).
2. J. Gosling et al., *The Java Language Specification, Java SE 21 Edition*, Chapter 17: Threads and Locks.
3. Oracle, *Java SE 21 API Documentation*: `java.lang.Thread`, `java.util.concurrent`, `java.util.concurrent.locks`, `java.util.concurrent.atomic`.
4. B. Goetz et al., *Java Concurrency in Practice*, Addison-Wesley, 2006.
5. P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson (Chapter 23: Concurrency).
6. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson (Chapter 12: Concurrency).
7. K. Sierra, B. Bates, and T. Gee, *Head First Java*, 3rd ed., O'Reilly (the chapters on threads and concurrency).
8. J. Bloch, *Effective Java*, 3rd ed., Addison-Wesley (Chapter 11: Concurrency, Items 78–84).

PART IV · CHAPTER 11

Database Programming

A program's variables and objects live in memory and disappear when the program ends. Student grades, bank accounts, and sensor readings, however, must be kept for years, found quickly, and changed safely by many programs at the same time. Databases do this job. The chapter begins with the idea of a database, how it differs from a file, and the relational model. It then turns to SQLite, an embedded database, and to SQL, the common language of relational databases. The SQL part shows how to create tables, insert, query, update, and delete records, and how to join and group tables. The following sections cover JDBC, Java's database interface, and the safe way to use user input. The last sections show how to turn rows into objects and how transactions run several statements as a single unit of work. All code was run with JDK 21 and the SQLite JDBC driver.

LEARNING OUTCOMES

Explain the concepts

Explain why a database is needed, how it differs from a flat file, and what the relational model (table, row, column, primary key) is.

Write SQL

Write `CREATE TABLE`, `INSERT`, `SELECT`, `UPDATE`, and `DELETE` statements; narrow a query with `WHERE`, `ORDER BY`, and `LIMIT`.

Use relationships

Link tables with a foreign key; combine them with `JOIN`; group rows with `GROUP BY` and `HAVING`; compare `NULL` values correctly.

Work with JDBC

Connect to SQLite from Java with `Connection`, `Statement`, and `ResultSet`, and insert, read, update, and delete records; close resources with try-with-resources.

Security and integrity

Prevent SQL injection with `PreparedStatement`; run several statements as one transaction with `commit` and `rollback`.

Map rows to objects
Turn the rows of a query result into objects and collect a table's SQL in a data access object (DAO).

11.1 The idea of a database

This section explains what a database is, which problem it solves, and how it organizes data. There is no code here; the concepts defined below underlie all of the JDBC code that follows.

11.1.1 Why do we need a database?

When a Java program executes `int total = 87;`, the value 87 is placed somewhere in RAM, the main memory. Objects live in the heap. This memory is fast but not persistent. When the program ends, the operating system reclaims its memory, and the variables and objects are gone. If the power fails, the contents of memory are erased as well.

Yet much data must outlive the program. A student's midterm grade is needed throughout the semester, a bank balance for years, a game's high score every time the game starts. Storing data this way is called **persistent storage**. Persistent storage lives on a medium such as a disk, which keeps its contents when the power is off.

Definition 11.1

A **database** is an organized collection of data that can be queried easily. The software that creates, protects, queries, and updates a database is called a **database management system** (DBMS).

In everyday speech the two are often confused: people say "the MySQL database", but MySQL is really a DBMS. The database is the set of tables and the records in them that the system manages. Figure 11.1 shows the difference between the two kinds of storage. While running, a program works on data in memory and writes whatever must last to the database (`INSERT` , `UPDATE`). When the program starts again, the data is read back from disk (`SELECT`).

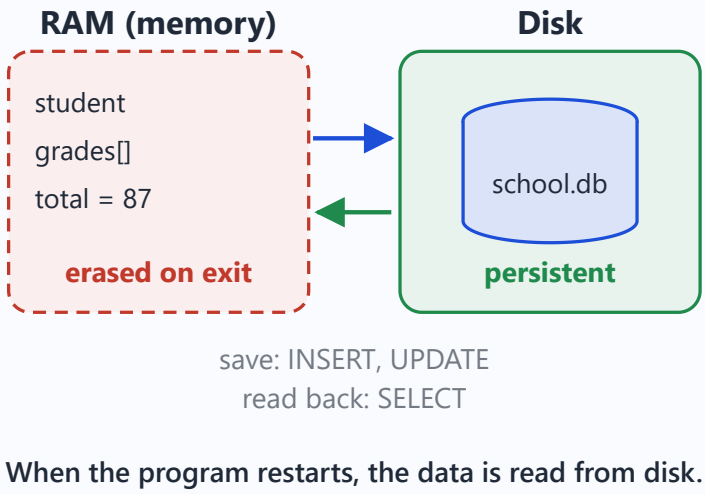


Figure 11.1: Memory and disk. Variables in RAM disappear when the program ends; the database file on disk (`school.db`) persists.

An analogy helps. A database is like a library's catalogued archive. Without a catalog, finding a book would mean walking past every shelf. With one, the book's location is found in a few steps. A database likewise finds the record you want among millions without reading all the data from beginning to end.

11.1.2 File or database?

A database is not the only way to store data persistently. A program can also write data to a text file or to a CSV (comma-separated values) file. For a settings file, a log, or a short list, a file is enough and simpler. As data grows and becomes more complex, and as several programs begin to use the same data, the limits of files appear. Table 11.1 compares the two approaches on six points.

Feature	Flat file (txt, csv)	Database
Search	The file is read from start to end	Fast with an index; a one-line SQL query
Update	Often the whole file is rewritten	Only the affected row changes
Concurrent access	Data can be corrupted if two programs write at once	Safe, thanks to locking and transactions
Consistency	Format checking is left entirely to the programmer	Data types and constraints (`NOT NULL` , `UNIQUE` , `CHECK`)
Security	File permissions: all or nothing	Permissions per user, table, and column (in server DBMSs; SQLite relies on file permissions)
After a failure	A write interrupted midway can corrupt the file	An interrupted transaction is rolled back

Table 11.1: Flat file versus database.

Each difference in the table corresponds to a concrete problem. If a school's student list is in a CSV file, finding one student among 20,000 rows means reading the file line by line. Changing one student's phone number usually means reading the whole file, changing it in memory, and writing it out again, because if a line in the middle of a text file grows longer, every byte after it shifts.

The most serious problem is concurrent access. If two teachers enter grades at the same time, both read the file, make their own changes, and write it back. The second one to write overwrites the first one's change without noticing it. A DBMS manages this situation with locks and transactions. If the power fails during a write, the file may be left half-written; a database rolls back the unfinished transaction at startup.

The difference in consistency is also large. Nothing stops someone from typing "twenty" instead of an age into a cell of a CSV file; every program that reads the file must check for this error itself. In a database the rules are part of the table definition, and a record that violates them is rejected up front. Figure 11.2 summarizes the structural difference: with a file, the program accesses the file directly; with a database, a DBMS sits in between.

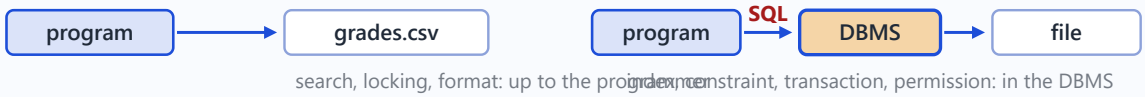


Figure 11.2: With a file, searching, locking, and format checking are left to the program; with a database, the program talks to the DBMS in SQL, and the DBMS handles indexes, locks, transactions, permissions, and type checking.

11.1.3 Core features

Four core features make databases superior to files.

Fast access to data. A database keeps extra structures called indexes for frequently searched columns. With an index, a record among millions is found quickly without looking at every row.

Column-level security. With a file, permissions usually apply to the whole file: whoever can read the file can read everything in it. A DBMS can grant permissions per table or even per column (field). In a company database, for example, only the accounting department may see the salary column, while other departments can read the name and phone columns of the same table. These permissions are granted in server DBMSs (for example with `GRANT` in PostgreSQL or MySQL); SQLite has no users or permissions, and access depends on file permissions.

Platform-independent, standard interfaces. SQL is a standard; its core statements are the same in all relational databases. On the Java side there is likewise a single set of interfaces, JDBC. The same Java code reaches different databases when the driver is swapped.

Independence from the file structure. The program does not know in which file, in what order, or in what format the data is stored on disk. Its job is to say what it wants in SQL; the DBMS decides how to find the data (Figure 11.3). Because of this separation, when the DBMS adds an index or changes how the data is laid out, the program code does not change.



Figure 11.3: The program says *what* it wants in SQL; the DBMS decides *how* to find it, using indexes and disk files.

Two more features come on top of these four. The first is integrity constraints: rules such as "this column cannot be empty", "this value cannot repeat", or "age cannot be negative" are written into the table, and the DBMS rejects any record that violates them. The second is transaction support: several changes are either all applied or none is.

11.1.4 The relational model

The most common kind of database today is the relational database. Its name comes from the fact that data is kept in tables based on the mathematical notion of a relation.

Definition 11.2

The **relational model** keeps data in tables made of rows and columns. A **column** is one attribute of the data; it has a name and a data type. A **row** (record) holds all the information about one entity. A **primary key** is a column that identifies each row of the table uniquely; no two rows can have the same primary key.

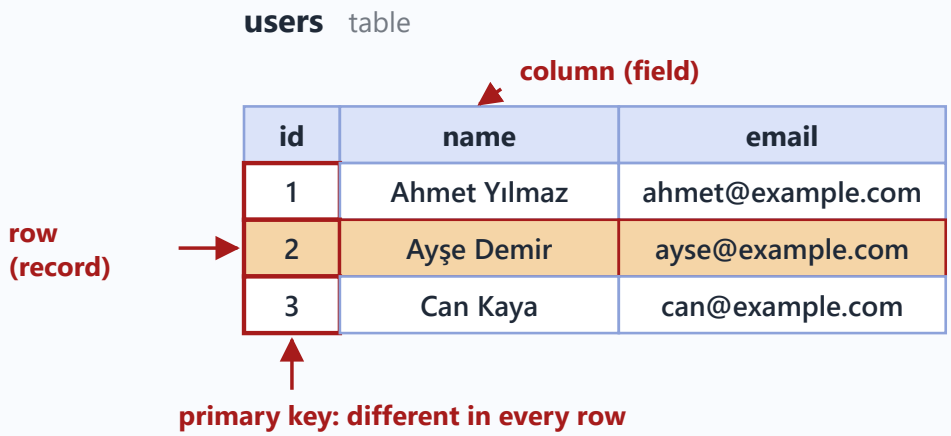


Figure 11.4: The `users` table. Its columns are `id`, `name`, and `email`; each row is one user; `id` is the primary key.

Figure 11.4 shows the `users` table used throughout the chapter. The table has three columns. The type of the `name` column is text (`TEXT`), and the type of the `id` column is integer (`INTEGER`). The highlighted second row holds everything about the user named Ayşe Demir: her ID is 2 and her e-mail is `ayse@example.com`.

Why is a primary key needed? Two users can have the same name; if two "Ahmet Yılmaz" records cannot be told apart, it is unclear which one to update. Even first and last name together are not unique. That is why each row is given a number that identifies it and nothing else. The number never changes and is never given to another row. The primary key is also used by other tables to point to this row.

An analogy with Java is useful. A **table corresponds to a class**, a **column to a field**, and a **row to an object**. The `users` table resembles a `User` class with the fields `int id`, `String name`, and `String email`; each row is an object of that class. The analogy is not exact: a table holds only data and has no methods. But this mapping is used directly when rows are converted to objects.

Common mistake

Confusing rows with columns. A column is an attribute found in every record (`email`). A row is one record and carries all of its attributes (Ayşe Demir's ID, name, and e-mail). "Adding a new student to the table" means adding a new row; "adding a birth date for students" means adding a new column.

11.1.5 Kinds of databases

Databases are grouped by the way they organize data. Three groups matter for this course.

Relational databases keep data in tables and are queried with SQL. Oracle, SQL Server, MySQL, and PostgreSQL belong to this group. These systems run as a server; many clients connect to the same database over the network. Most usage, about 77 percent, is in this group.

NoSQL and search systems do not use tables. They keep data as documents (JSON-like records), key-value pairs, or a search index. MongoDB is a document database and Elasticsearch is a search engine. They are used in particular scenarios, for example for data whose structure changes often or whose volume is very large. Their share of usage is about 23 percent (Figure 11.5).

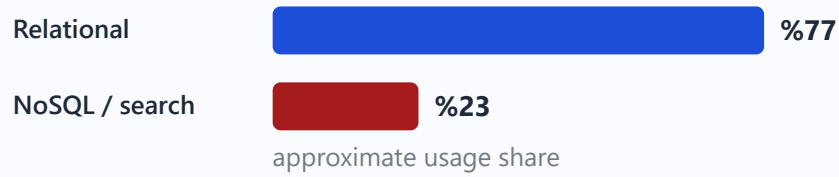


Figure 11.5: Approximate usage share: relational databases 77 percent, NoSQL and search systems 23 percent.

Embedded relational databases run not as a separate server but as a library inside the program. The best known is SQLite. They are common where little data is shared and a single program stores its own data: phone apps, browsers, embedded devices.

The examples in this chapter use SQLite, which is the easiest to set up. There is no server to install, no user to define, and no network to configure. The SQL statements and the JDBC code are almost the same in other relational databases. What you learn with SQLite carries over directly to MySQL or PostgreSQL; only the connection URL and the driver change.

11.2 SQLite

11.2.1 What is SQLite?

SQLite is an open-source, free relational database written in C. The following features set it apart from other relational systems.

- **It is embedded**: it runs as a library inside the program. When a Java program calls SQLite, no separate process starts; the database code runs in the program's own process.
- **It is serverless**: no separate server program, installation, or database administrator is needed.
- **It is self-contained**: all tables, indexes, and the schema are stored in a single disk file, for example `sample.db`. Backing up the database or moving it to another machine means copying this file.
- **It is cross-platform**: the same `.db` file opens on Windows, Linux, macOS, and Android.

SQLite is the most widely used embedded relational database in the world. It is found in every Android phone, every major browser, and many desktop programs. Common uses are mobile apps, embedded systems, small-scale applications, and desktop programs such as browsers and e-mail clients. For electrical and electronics engineering, embedded systems are especially important: a measuring device can store the values it reads in a SQLite file and query them later.

There is also a clear case where SQLite is not suitable: large systems in which many users write at the same time over a network. SQLite allows only one writer at a time. For the thousands of concurrent orders of an e-commerce site, a server-based system such as MySQL or PostgreSQL is needed.

11.2.2 Server-based and embedded databases

Figure 11.6 shows the difference between the two architectures. A server-based database involves two processes: the Java program and the database server. The program connects to the server over the network; MySQL's default port is 3306. A user name and password are supplied when connecting. The server writes the data to files on disk; the program never sees these files.

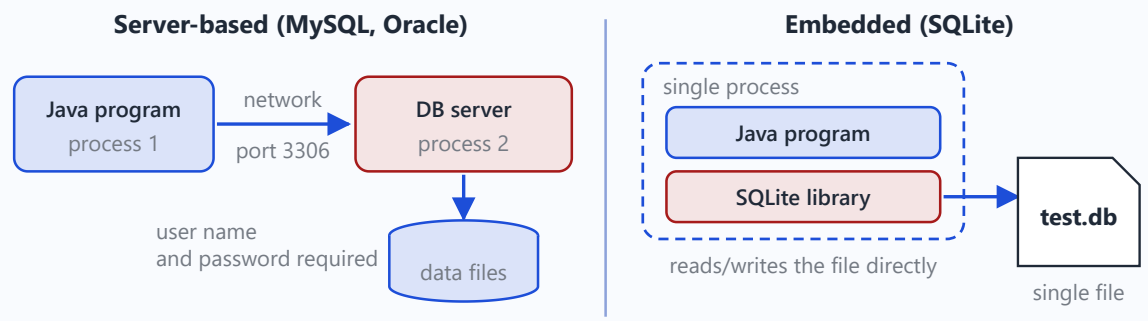


Figure 11.6: In a server-based database the program connects over the network to a separate server process; SQLite runs inside the program and writes directly to a single file.

An embedded database involves a single process. The SQLite library lives inside the Java program and reads and writes the `.db` file directly. There is no network, user name, or password. Table 11.2 summarizes the differences.

	Server-based	SQLite
Setup	Install a server; define users and passwords	Add a single jar to the project
Data	Many files managed by the server	A single `.db` file
Concurrent writes	Many users	One writer at a time

Table 11.2: A server-based database versus SQLite.

11.2.3 The driver, DB Browser, and running the code

For Java to talk to SQLite, a **JDBC driver** is needed. The common driver for SQLite is Xerial's `sqlite-jdbc` library (github.com/xerial/sqlite-jdbc). The driver is a single jar file, for example `sqlite-jdbc-3.40.0.0.jar`. This jar contains both the Java classes and SQLite's native library compiled for each operating system, so SQLite does not need to be installed separately.

To look inside the database file you can use **DB Browser for SQLite** (sqlitebrowser.org). This program opens a `.db` file, shows the tables and rows visually, and runs SQL statements. It is very useful for checking what a Java program has changed. After the program inserts a row, you refresh the table in DB Browser and see that the row was really added.

In Eclipse, the driver is added to the project as follows. Put the jar file in a `lib` folder in the project. Right-click the project and choose *Build Path → Configure Build Path…*. In the *Libraries* tab select *Classpath*, add the jar with *Add JARs…*, and close with *Apply and Close*. From the command line, compiling and running look like this:

```
javac Connect.java
java -cp ".;sqlite-jdbc-3.40.0.0.jar" Connect
```

The first line compiles the program. The driver is not needed for compiling, because the `java.sql` package that the program uses ships with the JDK. The second line runs the program. The `-cp` option gives the classpath: the dot is for `Connect.class` in the working folder, and the jar is for the driver. On Windows, classpath entries are separated by a semicolon (`;`); on Linux and macOS, by a colon (`:`). The driver is needed only at run time; if the jar is not on the classpath, the program compiles but fails when it tries to connect. The message in that case has the form "No suitable driver found …".

11.3 SQL basics

Before connecting to a database from Java, you need to know the database's language. A Java program sends SQL statements to the database as text; the database parses and runs them. The statements in this section work the same in DB Browser for SQLite and inside a Java program.

11.3.1 Common SQL statements

> **Definition 11.3**
>
> **SQL** (Structured Query Language) is the common language of relational databases. An SQL statement that asks the database for data is called a **query**.

Unlike Java, SQL is a declarative language. To search an array in Java you write a loop: you say step by step which element to look at and in what order. In SQL you describe only the result you want: "get the players whose score is above 70." The DBMS decides how to produce the result. Table 11.3 lists the five most commonly used statements.

Statement	Purpose	Example	CRUD
`CREATE TABLE`	Creates a new table	`CREATE TABLE users (id INTEGER, name TEXT)`	—
`INSERT INTO`	Inserts a new record	`INSERT INTO users(name) VALUES('Ali')`	**C**reate
`SELECT`	Queries a table	`SELECT * FROM users`	**R**ead
`UPDATE`	Updates existing records	`UPDATE users SET name = 'Efe' WHERE id = 1`	**U**pdate
`DELETE`	Deletes records	`DELETE FROM users WHERE id = 1`	**D**elete

Table 11.3: Common SQL statements and CRUD operations.

The four basic operations on persistent data are known as **CRUD**. The name comes from the initials of the operations: create, read, update, and delete. Almost every database application is a combination of these four operations. Figure 11.7 shows which SQL statement corresponds to each letter. `CREATE TABLE` is not part of CRUD; it creates the structure of the table, not data.



Figure 11.7: CRUD: inserting (`INSERT`), reading (`SELECT`), updating (`UPDATE`), and deleting (`DELETE`) rows of a table.

The syntax rules are short. SQL keywords are case-insensitive: `select` and `SELECT` are the same. For readability, keywords are written in upper case and table and column names in lower case. Text values go inside single quotes: `'Ali'`. Double quotes in SQL are used for names, not values; that is why SQL inside a Java string writes text values with single quotes, and the two languages do not get mixed up. The asterisk in `SELECT *` means "all columns". `WHERE` selects the rows to which the statement applies.

11.3.2 `CREATE TABLE`, data types, and constraints

A table is defined by its name and its columns. Each column has a name, a type, and optional constraints. The following statement creates a table of members:

```
CREATE TABLE IF NOT EXISTS members (
  id    INTEGER PRIMARY KEY,    -- unique
  name  TEXT NOT NULL,          -- required
  email TEXT UNIQUE,            -- no repeats
  age   INTEGER CHECK (age >= 0)
)
```

Each part of the statement has a job. `IF NOT EXISTS` prevents an error when the table already exists. The program can run this statement every time it starts: the first time the table is created, and later the statement does nothing. Parts that start with `--` are comments that run to the end of the line in SQL, like `//` in Java.

`INTEGER PRIMARY KEY` has a special meaning in SQLite. This column is the row's internal number (`rowid`). If no value is given on insert, SQLite assigns one more than the largest `id` in the table, or 1 if the table is empty. The program therefore does not need to generate numbers.

Constraints are rules written into the table. `NOT NULL` says the column cannot be left empty, `UNIQUE` says a value cannot repeat in two rows, and `CHECK` says the given condition must hold in every row. If a constraint is violated, the statement is rejected and the table does not change. The following experiment sends nine inserts to this table and prints the message JDBC gives for each rejected insert:

```
accepted: INSERT INTO members(name, email, age) VALUES ('Ada', 'ada@x', 20)
[SQLITE_CONSTRAINT_NOTNULL] A NOT NULL constraint failed (NOT NULL constraint failed:
members.name)
[SQLITE_CONSTRAINT_CHECK] A CHECK constraint failed (CHECK constraint failed: age >= 0)
[SQLITE_CONSTRAINT_UNIQUE] A UNIQUE constraint failed (UNIQUE constraint failed:
members.email)
[SQLITE_CONSTRAINT_PRIMARYKEY] A PRIMARY KEY constraint failed (UNIQUE constraint failed:
members.id)
accepted: INSERT INTO members(name) VALUES ('Naz')
accepted: INSERT INTO members(id, name) VALUES (10, 'Ece')
accepted: INSERT INTO members(name) VALUES ('Nur')
accepted: INSERT INTO members(name, age) VALUES ('Ali', 'twenty')
1 Ada ada@x 20 integer
2 Naz null null null
10 Ece null null null
11 Nur null null null
12 Ali null twenty text
```

Each message corresponds to one constraint. The row without a name is rejected with `NOT NULL constraint failed: members.name` , and the row with age −3 with `CHECK constraint failed: age >= 0` . The row that reuses Ada's e-mail runs into the `UNIQUE` constraint, and the second row with `id` 1 runs into the primary key. The `[SQLITE_CONSTRAINT_…]` part at the start of the message is an error code added by the driver; the actual message in parentheses comes from SQLite itself.

The last lines of the output show how `INTEGER PRIMARY KEY` behaves. Naz gets number 2. When Ece is given 10 by hand, the next row (Nur) gets 11: the largest `id` + 1. Numbers 3 to 9 stay unused; this is not a problem, because a primary key does not have to be consecutive, only unique.

The last row reveals a subtlety specific to SQLite: the text `'twenty'` could be stored in an `INTEGER` column. SQLite is flexibly typed; a column's type is only a suggestion (type affinity). If a value can be converted to a number it is stored as a number, and otherwise as text. `CHECK (age >= 0)` did not stop this row either, because in SQLite comparisons any text value counts as greater than any number. MySQL and PostgreSQL reject such a value. So on the Java side the value must be given with the right type, and numbers should be sent with typed methods such as `setInt`. Table 11.4 lists the Java counterparts of the SQLite types.

SQLite type	Java counterpart	Read method
`INTEGER`	`int`, `long`	`getInt`, `getLong`
`REAL`	`double`	`getDouble`
`TEXT`	`String`	`getString`
`BLOB`	`byte[]`	`getBytes`
`NULL`	no value	`getString` → `null`

Table 11.4: SQLite data types, their Java counterparts, and the methods that read them.

`BLOB` (binary large object) is a raw byte array; an image or a measurement record can be stored with this type. `NULL` is not a type but a special marker that indicates the absence of a value. How it behaves in comparisons is important in its own right.

11.3.3 SQL statements step by step

How the five basic statements change a table can be followed by running them in order on an empty database. The first statement creates the table:

```
CREATE TABLE users (id INTEGER PRIMARY KEY,
                    name TEXT, email TEXT)
```

After this statement the table has three columns but no rows. Two inserts follow:

```
INSERT INTO users(name, email)
  VALUES('Ahmet Yılmaz', 'ahmet@example.com')
```

```
INSERT INTO users(name, email)
  VALUES('Ayşe Demir', 'ayse@example.com')
```

The inserts do not give an `id`; only the `name` and `email` columns are named, and their values are given in the same order inside `VALUES`. SQLite assigns 1 to the first row and the largest `id` + 1 = 2 to the second. Each insert affects one row. Then the first row is updated and deleted, and finally the table is read:

```
UPDATE users SET name = 'Mehmet Akdeniz'
  WHERE id = 1
```

```
DELETE FROM users WHERE id = 1
```

```
SELECT * FROM users
```

If the statements are run one by one from a Java program and the table is printed after each, the result is Table 11.5. The "Rows affected" column is the count that JDBC reports for each statement.

Statement	Rows affected	State of the table
`CREATE TABLE users …`	0	(empty)
`INSERT … 'Ahmet Yılmaz'`	1	(1, Ahmet Yılmaz, ahmet@example.com)
`INSERT … 'Ayşe Demir'`	1	(1, Ahmet Yılmaz, ahmet@example.com) (2, Ayşe Demir, ayse@example.com)
`UPDATE … WHERE id = 1`	1	(1, **Mehmet Akdeniz**, ahmet@example.com) (2, Ayşe Demir, ayse@example.com)
`DELETE … WHERE id = 1`	1	(2, Ayşe Demir, ayse@example.com)
`SELECT * FROM users`	—	unchanged; result: (2, Ayşe Demir, ayse@example.com)

Table 11.5: The state of the `users` table after each statement (run with JDK 21 and SQLite JDBC).

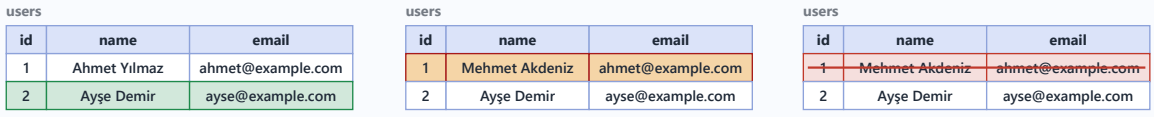


Figure 11.8: After the second insert (left), after the update (middle; only `name` changed), and after the delete (right; the row with number 1 is struck through).

An update changes only the column named in `SET`; `email` stays the same (Figure 11.8). `WHERE id = 1` selects only the first row. The delete uses the same condition and removes only the first row. The final `SELECT` does not change the table; it only reads and returns one row. The deleted number 1 is not passed to Ayşe: her `id` stays 2. A primary key does not change once it has been assigned.

Common mistake

Forgetting `WHERE` in `UPDATE` or `DELETE`. `UPDATE users SET name = 'X'` raises no error; it changes the name of *every* row in the table. `DELETE FROM users` deletes all rows but leaves the table itself. On a table with three rows, both statements return 3. The database does not treat this as an error; the only ways to undo it are to work inside a transaction or to restore a backup.

11.3.4 `SELECT`: filtering, sorting, limiting

`SELECT` is the most used and richest statement in SQL. The examples in this section work with a score table of five rows (Figure 11.9).

scores

player	score
ada	72
can	95
ece	58
efe	95
naz	81

Figure 11.9: The `scores` table.

A query consists of at most seven clauses; five of them are used here. `SELECT` says which columns to return, and `FROM` says which table to read. `WHERE` filters rows by a condition. The condition can use comparisons such as `=`, `<>` (not equal), `<`, and `>=`, and the connectives `AND` and `OR`. `LIKE 'e%'` means "starts with e": `%` stands for any number of characters and `_` for exactly one. `ORDER BY` sorts the result; `DESC` means descending, and the default `ASC` means ascending. `LIMIT n` takes the first `n` rows. Table 11.6 gives four queries and their results.

Query	Result
`SELECT player, score FROM scores WHERE score < 90 ORDER BY score DESC`	naz 81, ada 72, ece 58
`SELECT player, score FROM scores ORDER BY score DESC, player LIMIT 3`	can 95, efe 95, naz 81
`SELECT player, score FROM scores WHERE player LIKE 'e%'`	ece 58, efe 95
`SELECT COUNT(*), AVG(score), MAX(score) FROM scores WHERE score >= 60`	4 · 85.75 · 95

Table 11.6: Four queries on the `scores` table (results obtained by running them).

The first query selects scores below 90; can and efe (95) are dropped. The remaining three rows are sorted by score in descending order. The second query has no filter. Its sort has two keys: score descending first, and player name ascending when scores tie. can and efe tie at 95; can comes first alphabetically. `LIMIT 3` takes the first three rows. Without the second key, the order of tied scores would be undefined, and the result would depend on how the DBMS happened to lay out the data.

The third query returns the two names that start with e. In SQLite, `LIKE` is case-insensitive for ASCII letters; `LIKE 'E%'` gives the same two rows. The fourth query returns a single row, because `COUNT`, `AVG`, and `MAX` are **aggregate functions**: each produces one value from a set of rows. There are four scores of 60 or more: 72, 95, 95, and 81. Their sum is 343 and their average is 343 / 4 = 85.75. The maximum is 95. `SUM` and `MIN` belong to the same family.

Keep in mind how these functions behave on an empty set. With `WHERE score > 100`, where no row is selected, `COUNT(*)` returns 0, while `AVG(score)` and `SUM(score)` return `NULL`. The query still returns a single row; its values are empty.

11.3.5 Code trace: a `SELECT` query

An SQL statement does not run in the order it is written. The following Java code sends a query to the database and prints the result row by row. What matters here is not the JDBC types but the stages the query goes through.

```
String sql = "SELECT player, score FROM scores "
           + "WHERE score > 70 "
           + "ORDER BY player DESC LIMIT 2";
try (Statement st = conn.createStatement();
     ResultSet rs = st.executeQuery(sql)) {
  while (rs.next()) {
    System.out.println(rs.getString("player") + " "
                       + rs.getInt("score"));
  }
}
```

```
naz 81
efe 95
```

The database processes `FROM` first: it takes the five rows of the table. Then it applies the condition `WHERE score > 70` to each row and drops ece (58); four rows remain. Next, `ORDER BY player DESC` sorts the rows in reverse alphabetical order by name: naz, efe, can, ada. The sort is by name, not by score. Finally `LIMIT 2` takes the first two rows. The result set has two rows: (naz, 81) and (efe, 95). Figure 11.10 shows these stages.



Figure 11.10: Stages of the query. `WHERE` drops ece (left), `ORDER BY` sorts the names in reverse order (middle), and `LIMIT 2` makes the first two rows the result set (right).

The result reaches Java as a `ResultSet` object. This object has a **cursor**, which starts out *before* the first row. Each call to `rs.next()` advances the cursor by one row; it returns `true` if there is a row and `false` if there are none left. That is why the loop has the form `while (rs.next())`: the first call moves the cursor to the first row, and the body reads that row's columns. Table 11.7 gives the ten steps.

Step	Line	Variables	New output
1	1: `String sql = "SELECT player, score FROM scores "` 2: `+ "WHERE score > 70 "` 3: `+ "ORDER BY player DESC LIMIT 2";`	sql=SELECT … LIMIT 2	—
2	5: `ResultSet rs = st.executeQuery(sql)) {`	stage=FROM	—
3	5: `ResultSet rs = st.executeQuery(sql)) {`	stage=WHERE	—
4	5: `ResultSet rs = st.executeQuery(sql)) {`	stage=ORDER BY	—
5	5: `ResultSet rs = st.executeQuery(sql)) {`	stage=LIMIT	—
6	6: `while (rs.next()) {`	rs.next()=true	—
7	7: `System.out.println(rs.getString("player") + " "` 8: `+ rs.getInt("score"));`	player=naz, score=81	`naz 81`
8	6: `while (rs.next()) {`	rs.next()=true	—
9	7: `System.out.println(rs.getString("player") + " "` 8: `+ rs.getInt("score"));`	player=efe, score=95	`efe 95`
10	6: `while (rs.next()) {` 10: `}`	rs.next()=false	—

Table 11.7: Tracing the `SELECT` query. Steps 2 to 5 happen inside the database, steps 6 to 10 in the Java loop.

The logical order of execution is visible in this example: `FROM` → `WHERE` → `ORDER BY` → `LIMIT`. The column selection in `SELECT` happens near the end. This order matters even more in grouping queries. To predict what a query will return, read it in this order rather than in the order it is written.

Example 11.1

On the `scores` table, what do the queries `SELECT player FROM scores WHERE score > 90 OR player = 'ece'` and `SELECT player FROM scores LIMIT 2 OFFSET 1` return?

In the first, the condition is evaluated separately for each row. can and efe, with 95, satisfy the first part; ece satisfies the second. ada (72) and naz (81) satisfy neither. The result is **can, ece, efe**. Because no sort was requested, SQLite returns the rows in insertion order. In the second, `OFFSET 1` skips the first row and `LIMIT 2` takes the next two: **can, ece**. The queries gave these results when run. Without `ORDER BY`, the standard does not guarantee row order; if the order matters, always write `ORDER BY`.

11.4 Relationships and grouping

In a real application the data does not fit in a single table. Students, departments, courses, and orders live in separate tables and are linked to one another. This section explains why tables are split, how they are linked, and how linked tables are combined in a single query.

11.4.1 Normalization

Suppose a shop keeps its orders in a single table: each row holds the customer's name, phone, and the product ordered. If Ali places three orders, his name and phone are repeated in three rows (Figure 11.11, top). This repetition is not just wasted space; the real problem is the risk of inconsistency.

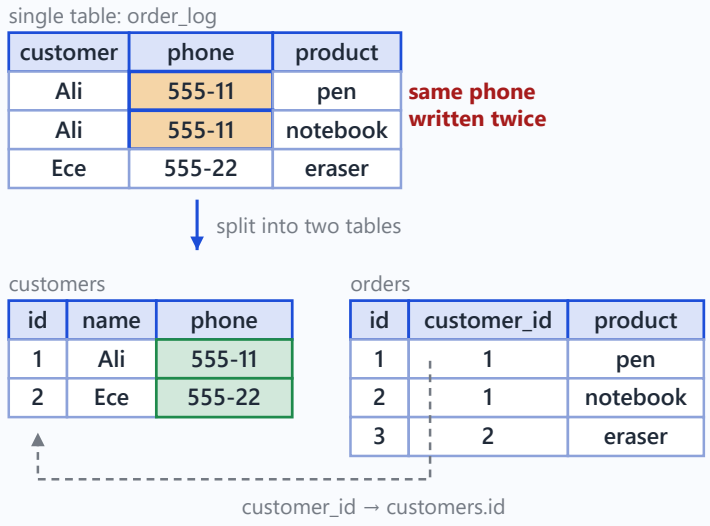


Figure 11.11: A single table with repeated data is split into two tables: `customers` and `orders`. Orders link to a customer through `customer_id`.

Repetition leads to three kinds of corruption, called **anomalies**. **Update anomaly**: when Ali's phone changes, every row must be corrected separately; if one is forgotten, Ali has two different phones and there is no way to know which is right. **Insert anomaly**: a customer who has no orders yet cannot be recorded, because every row is an order. **Delete anomaly**: when Ece's only order is deleted, Ece's phone is lost too.

Definition 11.4

Normalization is splitting data into tables so that each fact is stored in exactly one place, in order to avoid repetition. The split tables are linked to one another by a key.

In the lower part of the figure, the customer's information is stored once in the `customers` table. The `orders` table holds not the customer's name and phone but only the customer's number (`customer_id`). When Ali's phone changes, a single cell is updated. A customer with no orders can be added to `customers`. Even if Ece's order is deleted, Ece's record remains. All three anomalies disappear at once.

The first rule of normalization is called **first normal form**: each cell holds a single value. A customer's two phones are not written as a list in one cell such as `'555-11, 555-33'`. In such a cell, searching for "the customer whose phone is 555-33" becomes hard, and removing one number turns into text processing. The phones are moved either to separate rows or to a separate `phones(customer_id, number)` table. Higher normal forms exist as well; for this course the first rule and the principle "each fact in one place" are enough.

11.4.2 Foreign keys

The column that links split tables has a special name.

Definition 11.5

A **foreign key** is a column that points to another table's primary key. The value of a foreign key either equals the key of a row in that table or is `NULL`.

The chapter's relationship examples use two tables: `depts` and `students`. Every student has a department, and the link is made through the `dept_id` column:

```
CREATE TABLE depts (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL);
CREATE TABLE students (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  dept_id INTEGER REFERENCES depts(id));
```

`REFERENCES depts(id)` declares that `dept_id` is a foreign key. The following rows are inserted into the tables; Deniz's department is not yet known:

```
INSERT INTO depts(name) VALUES ('CS'), ('EE'), ('ME');
INSERT INTO students(name, dept_id)
  VALUES ('Ada', 1), ('Bora', 2), ('Cem', 1), ('Deniz', NULL);
```

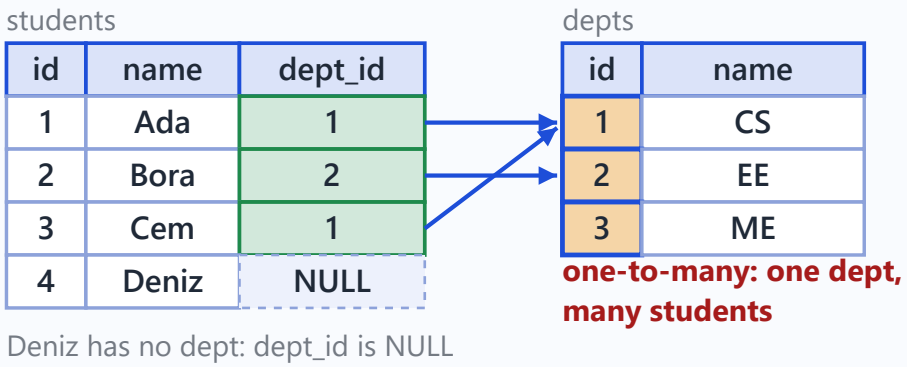


Figure 11.12: The `students.dept_id` column points to the `depts.id` column. A department can have many students: a one-to-many relationship. Deniz has no department; his `dept_id` is `NULL`.

Figure 11.12 shows the relationship. Ada and Cem both have `dept_id` 1; both are in the CS department. A department can have several students, but a student has only one department. This structure is called a **one-to-many** relationship. In Java the counterpart is that each `Student` object holds a `Dept` reference. In the table, a key value stands in place of the object reference.

The main benefit of a foreign key is integrity checking. Adding a student to the nonexistent department 9, or deleting a department that still has students, breaks the link: the student carries a number that points nowhere. A DBMS rejects both operations. SQLite, however, **does not perform this check by default**, for backward compatibility. The check must be enabled separately on every connection:

```
PRAGMA foreign_keys = ON;
INSERT INTO students(name, dept_id) VALUES ('Efe', 9);   -- rejected
DELETE FROM depts WHERE id = 2;                          -- rejected: Bora is linked
DELETE FROM depts WHERE id = 3;                          -- accepted: ME is empty
```

When the same operations are run first with the default setting and then with the check enabled, the following output is obtained:

```
== no PRAGMA (default)
foreign_keys = 0
accepted (1): INSERT INTO students(name, dept_id) VALUES ('Efe', 9)
accepted (1): DELETE FROM depts WHERE id = 2
== PRAGMA foreign_keys = ON
rejected: INSERT INTO students(name, dept_id) VALUES ('Efe', 9) ->
[SQLITE_CONSTRAINT_FOREIGNKEY] A foreign key constraint failed (FOREIGN KEY constraint
failed)
rejected: DELETE FROM depts WHERE id = 2 -> [SQLITE_CONSTRAINT_FOREIGNKEY] A foreign key
constraint failed (FOREIGN KEY constraint failed)
accepted (1): DELETE FROM depts WHERE id = 3
accepted (1): INSERT INTO students(name, dept_id) VALUES ('Nil', NULL)
```

With the default setting, `PRAGMA foreign_keys` is 0. The insert into department 9 is silently accepted; Bora's department is deleted too, and Bora's `dept_id` now points to a row that no longer exists. With the check enabled, both operations are rejected with `FOREIGN KEY constraint failed`. The ME department, which has no students, can be deleted. A `NULL` foreign key is always accepted, because it means "no department". In Java this setting is made right after the connection is opened, with `stmt.execute("PRAGMA foreign_keys = ON")`. The setting is specific to the connection: it must be repeated on every new connection.

Another kind of relationship besides one-to-many is also common: **many-to-many**. A student is enrolled in many courses, and a course has many students. This relationship cannot be built with columns of the two tables alone; a third, linking table is needed:

```
CREATE TABLE enrollments (
  student_id INTEGER REFERENCES students(id),
  course_id  INTEGER REFERENCES courses(id),
  PRIMARY KEY (student_id, course_id));
```

Each row of the `enrollments` table says "this student is enrolled in this course". Each of the two columns is a foreign key to a different table. The primary key is the combination of the two columns, so the same student cannot be enrolled in the same course twice.

11.4.3 Combining tables with `JOIN`

Normalization has a price: the answer to "Which department is Ada in?" is now spread over two tables. `students` holds only the number 1, and `depts` says that 1 is "CS". To bring the two pieces of information together in one result, the tables are joined.

Definition 11.6

A **join** (`JOIN`) is an SQL operation that matches the rows of two tables according to an `ON` condition and produces a single result table. A plain `JOIN` (or `INNER JOIN`) returns only the rows that have a match; a `LEFT JOIN` returns all rows of the left table.

```
SELECT s.name AS student, d.name AS dept
FROM students s JOIN depts d
  ON s.dept_id = d.id
ORDER BY s.id
```

In the query, `s` and `d` are table **aliases**: `FROM students s` says that `s` can be written in place of `students` in the rest of the query. Aliases are required here, because both tables have a column called `name` . `s.name` is the student's name and `d.name` is the department's name. `AS` gives a result column a new name; on the Java side the column is read by that name: `rs.getString("dept")` . `ON s.dept_id = d.id` is the matching condition: a student row is matched with the department row whose `id` equals its own `dept_id` .

Written with `LEFT JOIN` on the same tables, the query becomes:

```
SELECT s.name AS student, d.name AS dept
FROM students s LEFT JOIN depts d
  ON s.dept_id = d.id
ORDER BY s.id
```

The results of the two queries are side by side in Table 11.8. The only difference is Deniz. Because Deniz's `dept_id` is `NULL` , he matches no department. `JOIN` leaves a row without a match out of the result. `LEFT JOIN` keeps every row of the left table (`students`); if there is no match, it fills the right table's columns with `NULL` . In Java, reading this value with `getString` gives `null` .

`student`	`JOIN` (INNER)	`LEFT JOIN`
Ada	CS	CS
Bora	EE	EE
Cem	CS	CS
Deniz	no row	`NULL`

Table 11.8: Results of `JOIN` and `LEFT JOIN` .

Which one to choose depends on the question. To "list the students who have a department, with the department name", use `JOIN` ; to "list all students, and write the department name where there is one", use `LEFT JOIN` . The order of the tables matters in `LEFT JOIN` : if you write `depts d LEFT JOIN`

`students s`, the preserved side is the departments. That query returns the ME department with a `NULL` student.

Two mistakes are common. The first is writing `SELECT name` without an alias. SQLite cannot tell which `name` is meant and rejects the query with the message `ambiguous column name: name`. The second is forgetting the `ON` condition:

```
SELECT COUNT(*) FROM students, depts      -- 12
```

In a join without a condition, every row is matched with every row. Four students and three departments give 4 × 3 = 12 rows. This is called a **Cartesian product**. The result is meaningless: Ada appears in the CS, EE, and ME departments at once. With large tables this mistake is very expensive: 10,000 students and 100 departments produce a million rows.

11.4.4 Code trace: a `JOIN` query

The following code runs the join query from Java. `stmt` is a `Statement` created on an open connection.

```
String sql = "SELECT s.name AS student, d.name AS dept "
    + "FROM students s JOIN depts d "
    + "ON s.dept_id = d.id ORDER BY s.id";
try (ResultSet rs = stmt.executeQuery(sql)) {
  while (rs.next()) {
    System.out.println(rs.getString("student") + " - "
        + rs.getString("dept"));
  }
}
```

```
Ada - CS
Bora - EE
Cem - CS
```

For each row of `students`, the database looks in `depts` for the row whose `id` matches. Ada's `dept_id` is 1; department 1 is CS. Bora links to 2, that is, EE. Cem links to 1 again; the same department row is used a second time. CS is written once in the department table but appears twice in the result. Deniz's `dept_id` is `NULL`, and `NULL` is not considered equal to any `id`; Deniz does not enter the result (Figure 11.13).

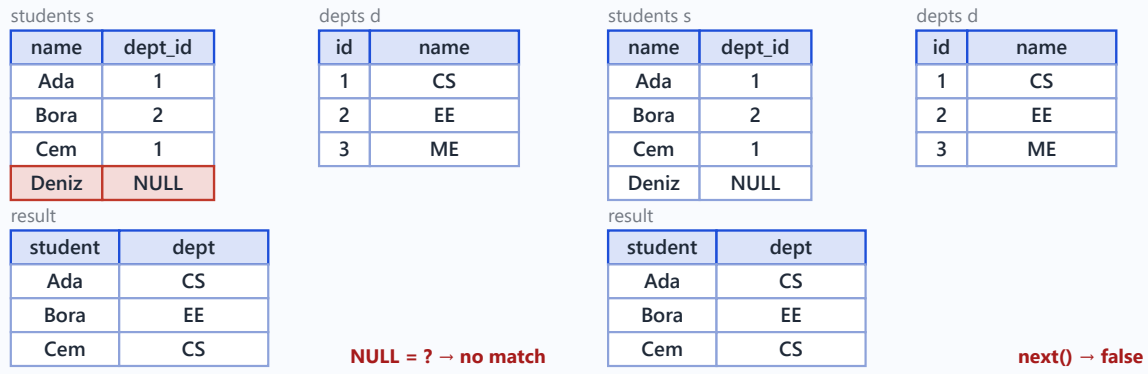


Figure 11.13: Deniz's `NULL` department matches no row (left); the result set has three rows (right).

Step	Line	Variables	New output
1	1: `String sql = "SELECT s.name AS student, d.name AS dept "` 2: `+ "FROM students s JOIN depts d "` 3: `+ "ON s.dept_id = d.id ORDER BY s.id";`	—	—
2	4: `try (ResultSet rs = stmt.executeQuery(sql)) {`	—	—
3	4: `try (ResultSet rs = stmt.executeQuery(sql)) {`	—	—
4	4: `try (ResultSet rs = stmt.executeQuery(sql)) {`	—	—
5	4: `try (ResultSet rs = stmt.executeQuery(sql)) {`	—	—
6	5: `while (rs.next()) {` 6: `System.out.println(rs.getString("student") + " - "` 7: `+ rs.getString("dept"));`	row=1	`Ada - CS`
7	5: `while (rs.next()) {` 6: `System.out.println(rs.getString("student") + " - "` 7: `+ rs.getString("dept"));`	row=2	`Bora - EE`
8	5: `while (rs.next()) {` 6: `System.out.println(rs.getString("student") + " - "` 7: `+ rs.getString("dept"));`	row=3	`Cem - CS`
9	5: `while (rs.next()) {` 9: `}`	row=none	—

Table 11.9: Tracing the `JOIN` query.

The Java side is the same as in the previous example: the cursor advances row by row through the result set. The columns are read by the names given with `AS`: `"student"` and `"dept"`. If `rs.getString("name")` had been written, the driver would raise an error, because the result has no column called `name`. With `LEFT JOIN`, `Deniz - null` would appear as a fourth line; this was verified by running it.

11.4.5 `GROUP BY` and `HAVING`

Aggregate functions give a single value when applied to a whole table. Often, though, you want a separate value for each group: the number of orders per customer, the number of students per department. The examples in this section use an order table of six rows:

```
INSERT INTO orders(customer, amount) VALUES
  ('Ali', 120), ('Ece', 80), ('Ali', 50),
  ('Can', 200), ('Ali', 30), ('Ece', 40);
```

`GROUP BY customer` gathers the rows that have the same `customer` value into one group. Aggregate functions are computed separately for each group, and each group becomes a single row in the result (Figure 11.14). Ali has three orders: 120 + 50 + 30 = 200. Can has a single order of 200, and Ece has two orders: 80 + 40 = 120.

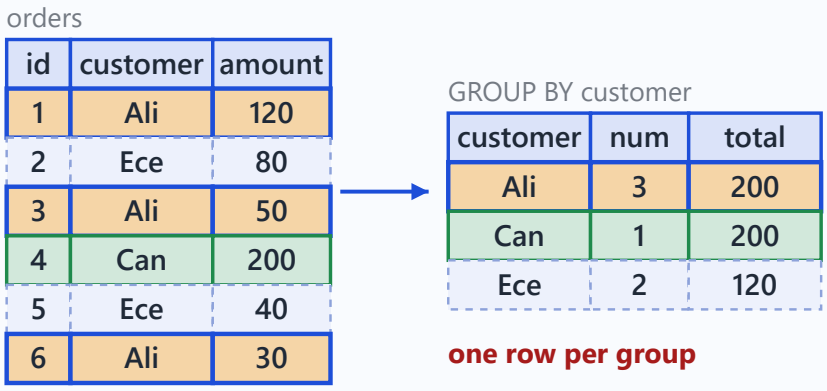


Figure 11.14: The `orders` table is split into three groups by customer; the count and the total are computed for each group.

Filtering can happen before or after grouping. `WHERE` filters rows *before* grouping. `HAVING` filters groups *after* grouping. That is why a condition containing an aggregate function cannot go in `WHERE`: when `WHERE` runs there are no groups yet, and therefore no `SUM(amount)`. If you write `WHERE SUM(amount) >= 150`, SQLite rejects it with the message `misuse of aggregate: SUM()`. Table 11.10 compares three cases.

Rest of the query	Result
`GROUP BY customer`	Ali (3, 200) · Can (1, 200) · Ece (2, 120)
`WHERE amount >= 50 GROUP BY customer`	Ali (2, 170) · Can (1, 200) · Ece (1, 80)
`GROUP BY customer HAVING SUM(amount) >= 150`	Ali (3, 200) · Can (1, 200)

Table 11.10: Three forms of the query `SELECT customer, COUNT(*), SUM(amount) FROM orders …`.

In the second row, `WHERE amount >= 50` first filters the individual orders: Ali's order of 30 and Ece's order of 40 are dropped. The groups are built from the remaining orders. Ali's count falls to 2 and his total to 170. In the third row, all orders are grouped first, and then Ece's group, whose total is below 150, is dropped. The two queries look alike but answer different questions: one says "count the orders of 50 TL or more by customer", the other "find the customers whose total is 150 or more".

The clauses of a query run logically in this order:

> `FROM` → `WHERE` → `GROUP BY` → `HAVING` → `SELECT` → `ORDER BY` → `LIMIT`

This order explains many rules. `WHERE` cannot use aggregate functions, because it comes before grouping. `ORDER BY` can use aliases defined in `SELECT`, because it comes after it.

Grouping is also used together with joins. The number of students in each department is found like this:

```
SELECT d.name, COUNT(s.id)
FROM depts d LEFT JOIN students s
  ON s.dept_id = d.id
GROUP BY d.id
```

The result is CS 2 · EE 1 · ME 0. Two details matter here. Thanks to `LEFT JOIN`, ME, which has no students, also appears in the result; a plain `JOIN` would have hidden it. The second detail is `COUNT(s.id)`. `COUNT(column)` counts only values that are not `NULL`. In ME's single joined row, `s.id` is `NULL` and is not counted; the result is 0. `COUNT(*)`, on the other hand, counts rows. Had it been used, it would have given ME a 1; this was verified by running it.

There is one more trap in grouping. In the query `SELECT customer, amount FROM orders GROUP BY customer`, `amount` is neither grouped nor inside an aggregate function. Ali's group has three different amounts; which one should be shown? SQLite raises no error and returns the value from one of the rows in the group. In a test, Ali came out as 120, Can as 200, and Ece as 80. Many databases, such as PostgreSQL and SQL Server, reject this query. The rule is simple: every column in `SELECT` either appears in `GROUP BY` or is inside an aggregate function.

11.4.6 Code trace: a `GROUP BY` query

The following code combines grouping, group filtering, and a two-key sort in a single query:

```
String sql = "SELECT customer, COUNT(*) AS num, "
    + "SUM(amount) AS total FROM orders "
    + "GROUP BY customer HAVING SUM(amount) >= 150 "
    + "ORDER BY total DESC, customer";
try (ResultSet rs = stmt.executeQuery(sql)) {
  while (rs.next()) {
    System.out.println(rs.getString("customer") + ": "
        + rs.getInt("num") + " orders, "
        + rs.getInt("total") + " TL");
  }
}
```

```
Ali: 3 orders, 200 TL
Can: 1 orders, 200 TL
```

The database first splits the six rows into three groups by customer. For each group it computes `COUNT(*)` and `SUM(amount)`: Ali 3 and 200, Can 1 and 200, Ece 2 and 120. `HAVING SUM(amount) >= 150` drops Ece's group. `ORDER BY total DESC, customer` sorts the remaining two groups by total in descending order. Ali and Can tie at 200; since the second key is the name, Ali comes first (Figure 11.15).

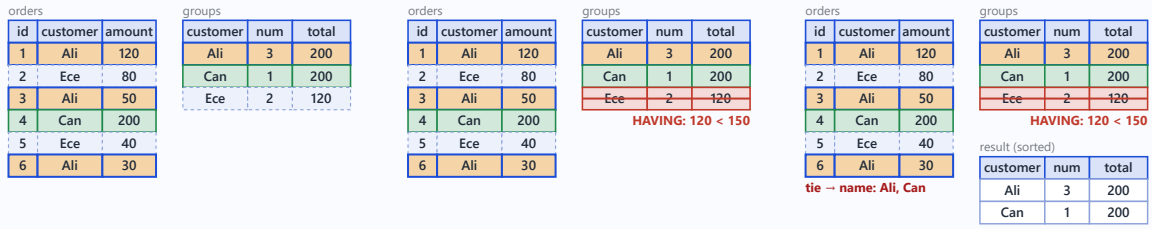


Figure 11.15: The groups and their totals (left), `HAVING` drops Ece (middle), the sorted result (right).

Step	Line	Variables	New output
1	1: `String sql = "SELECT customer, COUNT(*) AS num, "` 2: `+ "SUM(amount) AS total FROM orders "` 3: `+ "GROUP BY customer HAVING SUM(amount) >= 150 "` 4: `+ "ORDER BY total DESC, customer";`	—	—
2	5: `try (ResultSet rs = stmt.executeQuery(sql)) {`	—	—
3	5: `try (ResultSet rs = stmt.executeQuery(sql)) {`	—	—
4	5: `try (ResultSet rs = stmt.executeQuery(sql)) {`	—	—
5	5: `try (ResultSet rs = stmt.executeQuery(sql)) {`	—	—
6	6: `while (rs.next()) {` 7: `System.out.println(rs.getString("customer") + ": "` 8: `+ rs.getInt("num") + " orders, "` 9: `+ rs.getInt("total") + " TL");`	row=1	`Ali: 3 orders, 200 TL`

Step	Line	Variables	New output
7	6: `while (rs.next()) {` 7: `System.out.println(rs.getString("customer") + ": "` 8: `+ rs.getInt("num") + " orders, "` 9: `+ rs.getInt("total") + " TL");`	row=2	`Can: 1 orders, 200 TL`
8	6: `while (rs.next()) {` 11: `}`	row=none	—

Table 11.11: Tracing the `GROUP BY` query.

On the Java side, the aliases (`num` , `total`) are read like column names: `rs.getInt("num")` . The result has one row per group; the individual orders are not visible. Ali's three orders have become a single line, "Ali: 3 orders, 200 TL".

11.4.7 `NULL` : the absence of a value

Deniz's department is unknown. This is shown in the table as `NULL` .

> **Definition 11.7**
>
> In SQL, `NULL` is a special marker that means a value is unknown or absent. `NULL` is neither 0 nor the empty string (`''`). Every comparison involving `NULL` gives the result "unknown".

SQL uses a three-valued logic: true, false, and unknown. The comparison `dept_id = NULL` is not true even for Deniz; it is unknown. Whether an unknown value equals another unknown value is unknown as well. `WHERE` selects only the rows whose condition is *true*; that is why the first query below returns no rows:

```
SELECT name FROM students WHERE dept_id = NULL;      -- 0 rows
SELECT name FROM students WHERE dept_id IS NULL;     -- Deniz
SELECT name FROM students WHERE dept_id <> 1;        -- only Bora
SELECT name, COALESCE(dept_id, 0) FROM students;     -- Deniz 0
SELECT COUNT(*), COUNT(dept_id) FROM students;     -- 4 and 3
```

The correct forms are `IS NULL` and `IS NOT NULL` . The result of the third line is more surprising. `dept_id <> 1` does not select Deniz either, because it is not even known whether `NULL` differs from 1. If you want "the students outside CS", write `WHERE dept_id <> 1 OR dept_id IS NULL` .

`NULL` also propagates through arithmetic: the result of `NULL + 1` is `NULL` . `COALESCE(dept_id, 0)` returns the second argument if the first is `NULL` ; this puts 0 in place of `NULL` . Aggregate functions skip `NULL` s. `COUNT(*)` counts rows and gives 4; `COUNT(dept_id)` counts the values that are not `NULL` and gives 3. If no rows match, `AVG` and `SUM` return `NULL` and `COUNT` returns 0. All of these lines were verified by running them in a Java program.

There is a problem on the Java side as well. Primitive types such as `int` cannot be `null`. `rs.getInt` returns 0 for a `NULL` column. But is the 0 really 0, or was it `NULL`? To tell the difference, call `wasNull()` right after the read:

```
String sql = "SELECT name, dept_id FROM students "
    + "WHERE id >= 3";
try (ResultSet rs = stmt.executeQuery(sql)) {
  while (rs.next()) {
    int d = rs.getInt("dept_id");
    boolean none = rs.wasNull();
    System.out.println(rs.getString("name") + " " + d
        + (none ? " (NULL)" : ""));
  }
}
```

```
Cem 1
Deniz 0 (NULL)
== late wasNull
Cem 1
Deniz 0
getString: null, getObject: null
SELECT name FROM students WHERE dept_id = NULL
  -> 0 rows:
SELECT name FROM students WHERE dept_id IS NULL
  -> 1 rows: Deniz;
SELECT name FROM students WHERE dept_id IS NOT NULL
  -> 3 rows: Ada; Bora; Cem;
SELECT name FROM students WHERE dept_id <> 1
  -> 1 rows: Bora;
SELECT NULL + 1, NULL = NULL, NULL IS NULL, COALESCE(NULL, 0)
  -> 1 rows: null null 1 0;
SELECT name, COALESCE(dept_id, 0) FROM students
  -> 4 rows: Ada 1; Bora 2; Cem 1; Deniz 0;
SELECT COUNT(*), COUNT(dept_id) FROM students
  -> 1 rows: 4 3;
SELECT AVG(dept_id), SUM(dept_id) FROM students WHERE id > 10
  -> 1 rows: null null;
SELECT AVG(dept_id), SUM(dept_id) FROM students
  -> 1 rows: 1.33333333333333 4;
SELECT name FROM students WHERE NOT (dept_id = 1)
  -> 1 rows: Bora;
```

The first two lines of the output come from the program above. Cem's department is 1, and `wasNull()` returns `false`. For Deniz, `getInt` gives 0, `wasNull()` returns `true`, and "(NULL)" is appended to the line. `wasNull()` looks only at the *last column read*. In the second experiment, labeled "late wasNull", `getString("name")` is read after `dept_id`, and `wasNull()` is called after that. This time the name is what was asked about; since the name is not `NULL`, "(NULL)" is not

written on Deniz's line. `getString` and `getObject`, on the other hand, return Java's own `null` directly for `NULL`; `wasNull()` is not needed with these methods. The remaining lines of the output are the actual results of the SQL queries above.

> **Common mistake**
>
> Writing `WHERE dept_id = NULL`. The query raises no error; it just returns no rows, and the program concludes that "there are no students without a department". A check for `NULL` is always written with `IS NULL`. The Java counterpart is similar: `getInt` returning 0 does not show that the value is 0; `wasNull()` is called right after the relevant `getInt`.

11.4.8 Indexes

The secret of a database's fast search is the index.

> **Definition 11.8**
>
> A **database index** is an additional structure that keeps the values of a column in sorted order and points to the row where each value is found. SQLite stores an index as a B-tree.

An index is like the index at the back of a book. The pages where the word "inheritance" appears can be found from the index without reading the book from cover to cover. Without an index, the query `WHERE name = 'Cem'` looks at every row of the table; this is called a **scan**. With an index, a search similar to binary search is made in the sorted structure, and the database goes directly to Cem's row (Figure 11.16). In a table of n rows, a scan needs n comparisons, while a search in the B-tree takes about $\log n$ steps. For a million rows, that is the difference between a million and about twenty.

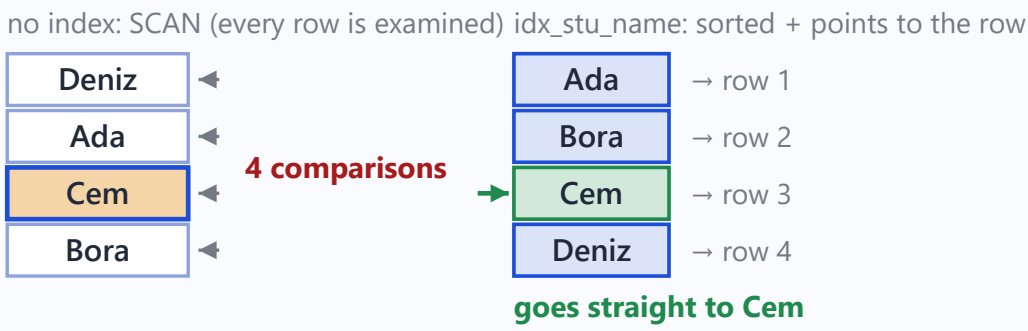


Figure 11.16: Without an index, all rows are scanned (4 comparisons); the index is sorted, so it goes straight to the name being searched.

An index is created with `CREATE INDEX`. SQLite's `EXPLAIN QUERY PLAN` statement shows how a query will be executed, that is, whether the table will be scanned or an index will be used:

```
CREATE INDEX idx_stu_name ON students(name);
EXPLAIN QUERY PLAN
  SELECT * FROM students WHERE name = 'Cem';
```

Table 11.12 gives the plans for four cases. The plans were obtained with SQLite 3.40.0.

Case	`EXPLAIN QUERY PLAN` output
Without an index, `WHERE name = 'Cem'`	`SCAN students`
With an index, `WHERE name = 'Cem'`	`SEARCH students USING INDEX idx_stu_name (name=?)`
`WHERE id = 3`	`SEARCH students USING INTEGER PRIMARY KEY (rowid=?)`
`WHERE name LIKE '%em'`	`SCAN students`

Table 11.12: `EXPLAIN QUERY PLAN` output.

`SCAN` shows that the table will be scanned, and `SEARCH` that an index will be used for the search. No separate index is needed for a primary key; `INTEGER PRIMARY KEY` is already the table's own ordering key. The index of a `UNIQUE` column is created automatically too, because the DBMS needs it to check for duplicates.

The last row shows the limit of an index. `LIKE '%em'` means "ends with em". An index keeps names sorted from the beginning; names that end in "em" are scattered throughout the sequence. A leading wildcard therefore cannot use the index. In the test, `LIKE 'C%'` also produced a `SCAN`. In SQLite `LIKE` is case-insensitive, while an ordinary index is case-sensitive, so even a prefix search did not use the index with default settings.

An index has a cost. Every `INSERT`, `UPDATE`, and `DELETE` updates not only the table but also the index, and the index takes up disk space. That is why not every column gets an index. An index goes on columns that are searched often and used in `WHERE`, `JOIN`, or `ORDER BY`. To see how large the effect is, 200 searches by the `name` column were run on a table of 200,000 rows. Without an index the searches took 633 ms; after the index was created, the same searches took 1 ms. The times vary with the machine and the run; the ratio is hundreds of times each time.

11.5 JDBC

SQL statements are the database's language; the Java program must pass these statements to the database and get the results back. JDBC builds this bridge.

11.5.1 What is JDBC?

Definition 11.9

JDBC (Java Database Connectivity) is the standard API (application programming interface) that Java offers for connecting to databases. It lives in the `java.sql` package and ships with the JDK. The code that talks to a particular database is supplied by that database's **JDBC driver**.

The structure of JDBC is a fine application of the interface concept from object-oriented programming. `Connection`, `Statement`, and `ResultSet` in the `java.sql` package are interfaces; they only say which methods will exist. The driver of each database vendor implements these interfaces with its own classes. The SQLite driver knows how to talk to a SQLite file, and the MySQL driver knows how to talk to a MySQL server (Figure 11.17).

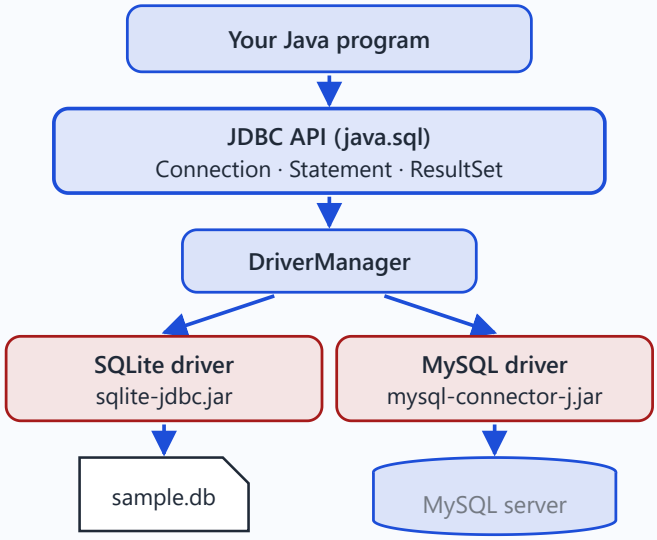


Figure 11.17: The JDBC layers. The program uses only the JDBC interfaces; `DriverManager` picks the right driver; the driver connects to a SQLite file or a MySQL server.

Because the program knows only the interfaces, switching databases often means changing only the driver jar and the connection URL. Polymorphism is at work here. The type of the variable in the program is `Connection`. At run time, the class of the actual object the variable refers to is `org.sqlite.jdbc4.JDBC4Connection` in the SQLite driver. This name was verified by printing `conn.getClass().getName()`. When the program calls `conn.createStatement()`, the JVM runs this class's implementation of the `Connection` interface. The same line runs the MySQL driver's implementation with MySQL.

An object of an interface cannot be created directly. If you write `new Connection()`, the compiler rejects it:

```
Main.java:4: error: Connection is abstract; cannot be instantiated
    Connection conn = new Connection();
                      ^
1 error
```

It is possible to build the connection object with `new` from the driver's class, but that would be wrong, because the program would become dependent on one driver. A connection is always obtained with `DriverManager.getConnection`; the driver decides which class's object is returned.

11.5.2 The core JDBC types

A JDBC program is written with a few types. Table 11.13 lists them and their important methods.

Type	Purpose	Important methods
`DriverManager`	Finds the right driver and opens a connection	`getConnection(url)`, `getConnection(url, user, password)`
`Connection`	An open session with the database	`createStatement()`, `prepareStatement(sql)`, `commit()`, `rollback()`, `close()`
`Statement`	Runs a fixed SQL text	`execute`, `executeUpdate`, `executeQuery`
`PreparedStatement`	Pre-compiled SQL with `?` placeholders	`setInt`, `setString`, `executeUpdate()`, `executeQuery()`
`ResultSet`	The rows of a query result and a cursor	`next()`, `getInt("id")`, `getString("name")`
`SQLException`	A database error; a checked exception	`getMessage()`, `getErrorCode()`

Table 11.13: The core JDBC types. All are in the `java.sql` package.

The types form a chain. `DriverManager` gives a `Connection`. The `Connection` produces a `Statement` or a `PreparedStatement`. When that object runs a query, a `ResultSet` comes back. Each link is obtained from the one before it; the programmer calls only `DriverManager` directly.

To use these types, write `import java.sql.*;` at the top of the file. If it is forgotten, the compiler cannot find the types:

```
Main.java:3: error: cannot find symbol
    Connection conn = DriverManager.getConnection("jdbc:sqlite:sample.db");
    ^
  symbol:   class Connection
  location: class Main
Main.java:3: error: cannot find symbol
    Connection conn = DriverManager.getConnection("jdbc:sqlite:sample.db");
                      ^
  symbol:   variable DriverManager
  location: class Main
2 errors
```

`SQLException` is a checked exception. Almost all JDBC methods declare that they may throw it. JDBC code therefore either sits inside a `try`-`catch` or the method declares `throws SQLException`. If neither is done, the program does not compile:

```
Main.java:4: error: unreported exception SQLException; must be caught or declared to be
thrown
    Connection conn = DriverManager.getConnection("jdbc:sqlite:sample.db");
                                                  ^
1 error
```

`Connection`, `Statement`, and `ResultSet` hold resources in the database: an open file, locks, memory. All three implement the `AutoCloseable` interface and must be closed when you are done with them. The safest way is try-with-resources.

11.5.3 The connection URL

`DriverManager` learns which database to connect to from a text address. This address is called the **JDBC URL** (Figure 11.18).

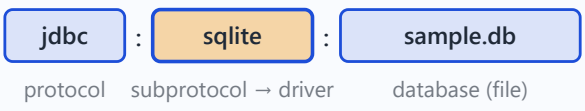


Figure 11.18: The parts of a JDBC connection URL: the protocol (`jdbc`), the subprotocol (selects the driver), and the file or address of the database.

The format of the URL is `jdbc:<subprotocol>:<database file or address>`. The first part is always `jdbc`. The second part determines the driver: `sqlite`, `mysql`, `postgresql`. `DriverManager` asks the registered drivers in turn whether they accept the URL. The first driver that accepts it opens the connection. The format of the third part is specific to the driver. Table 11.14 gives examples.

Connection URL	Meaning
`jdbc:sqlite:sample.db`	`sample.db` in the working folder; created empty if it does not exist
`jdbc:sqlite:C:/data/school.db`	A file given by its full path
`jdbc:sqlite::memory:`	A temporary in-memory database; deleted when the connection closes
`jdbc:mysql://localhost:3306/testdb`	A MySQL server: machine, port, database name; a user name and password are also needed

Table 11.14: Examples of connection URLs.

An in-memory database is very handy for experiments: it leaves no file on disk and starts clean on every run. Many examples in this chapter were run with `jdbc:sqlite::memory:`. However, an in-memory database belongs to its connection. A table created in one connection does not exist in a second connection opened with the same URL. In the test, the second connection got the message `no such table: t`.

A relative file path is resolved against the working folder. In Eclipse the working folder is the project folder; on the command line it is the folder from which the `java` command was given. So if a program is run from two places, two separate `sample.db` files can appear. SQLite creates a file that does not exist but does not create a folder that does not exist. With the URL `jdbc:sqlite:sub/folder/school.db`, if the folder `sub` does not exist, the connection fails with the message "path … does not exist".

In older books and code, the following line appears before the connection: `Class.forName("org.sqlite.JDBC");`. This line loaded the driver class and had it register itself with `DriverManager`. Since JDBC 4.0 (Java 6), the driver is loaded automatically from the jar on the classpath, and the line is no longer needed. Writing it does no harm. What happens if the driver is not on the classpath? When the program was run without the driver jar, this output was obtained:

```
No suitable driver found for jdbc:sqlite:sample.db
ClassNotFoundException: org.sqlite.JDBC
```

The first line is the message of the `SQLException` thrown by `getConnection`: no driver that accepts the URL was found. The same message appears when the URL has a typo. For example, the URL `jdbc:sqlit:school.db` gives this message even when the driver is loaded, because no driver recognizes the subprotocol `sqlit`. The second line is the `ClassNotFoundException` received when the old method looks up the driver class.

11.5.4 The steps of a JDBC program

Every JDBC program follows the same five steps (Figure 11.19): open a connection, create a statement object, run the SQL, process the result, and close the resources.

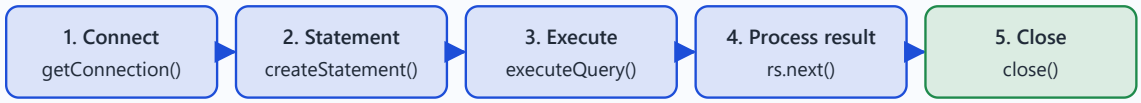


Figure 11.19: The five steps of a JDBC program: connect, create a statement, execute, process the result, close.

```
try (Connection conn = DriverManager.getConnection(url); // 1
     Statement stmt = conn.createStatement();             // 2
     ResultSet rs = stmt.executeQuery(sql)) {             // 3
  while (rs.next()) {                                     // 4
    System.out.println(rs.getInt(1));
  }
} // 5: rs, stmt, conn auto-closed in reverse
```

The structure of the code matches the steps one to one. Steps one, two, and three happen in order in the try-with-resources parentheses: the connection is opened, a `Statement` is taken from it, and the `Statement` runs the query and returns a `ResultSet`. Step four is the `while (rs.next())` loop. No code is written for step five: when the block ends, even after an error, the resources are closed in the reverse order of opening. First `rs`, then `stmt`, and last `conn`.

The reverse order is not arbitrary. A `ResultSet` depends on its `Statement`, and the `Statement` depends on its `Connection`. If the connection were closed first, the others would be left dangling. To verify the closing, the resources were copied into variables outside the block and queried with `isClosed()` after the block ended:

```
7
8
true true true
The database has been closed
```

All three resources are closed (`true true true`). An attempt to run a query on the closed `Statement` threw a `SQLException`; its message is on the last line. Connections that are not closed can leave the SQLite file locked. Another program, or even another connection of the same program, cannot write to the file. In long-running programs, objects that are not closed also cause memory leaks.

The code has one more detail: `rs.getInt(1)`. Columns can be read by name or by position. In JDBC, column numbers start **at 1**. For a programmer used to Java arrays starting at 0, this is a frequent source of errors.

Common mistake

Trying to use a try-with-resources resource outside the block. The scope of a variable declared in the parentheses is the block only; when the block ends, the resource is both closed and out of sight. The compiler rejects the use outside:

```
Main.java:6: error: cannot find symbol
    conn.close();
    ^
  symbol:   variable conn
  location: class Main
1 error
```

11.6 Basic operations

The six small programs in this section run in sequence on the same file, `sample.db`: connecting, creating a table, inserting, reading, updating, and deleting. The programs are independent; each opens and closes its own connection. The data stays persistently in the file between programs. Each program was run in full inside `public static void main` with `import java.sql.*;`; the chapter shows only the body of `main`.

11.6.1 Connecting to SQLite

```
// Path to the database file (connection URL)
String url = "jdbc:sqlite:sample.db";
try (Connection conn = DriverManager.getConnection(url)) {
  // Print a message if connected.
  if (conn != null) {
    System.out.println("Connected!");
  }
} catch (SQLException e) {
  // Print the error message on failure.
  System.out.println(e.getMessage());
}
```

```
Connected!
```

The program keeps the connection URL in a variable and opens the connection with `getConnection`. If the file does not exist, SQLite creates it; `sample.db`, which did not exist before the first run, is on disk after it. One interesting detail: the size of the created file is 0 bytes. SQLite writes no content to

the file until the first table is written. This was seen by asking a `File` object about the file's existence and size.

The `if (conn != null)` check is only an extra safeguard. If `getConnection` fails, it does not return `null`; it throws `SQLException`. So if execution reaches the inside of the `try` block, the connection is certainly open. The `catch` block prints the error message; if the driver were not on the classpath, it would print "No suitable driver found ..." here.

11.6.2 Creating a table

```
String url = "jdbc:sqlite:sample.db";
String sql = "CREATE TABLE IF NOT EXISTS users "
           + "(id INTEGER PRIMARY KEY, name TEXT, email TEXT)";
try (Connection conn = DriverManager.getConnection(url);
     Statement stmt = conn.createStatement()) {
  // Create the table.
  stmt.execute(sql);
  System.out.println("Table created!");
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

```
Table created!
```

This time there are two resources in the try-with-resources parentheses: the connection and a `Statement`. Resources inside the parentheses are separated by semicolons. `execute(sql)` runs any kind of SQL. Its return value says whether the statement produced a result set. `CREATE TABLE` produces no result set; `execute` returns `false`. For `SELECT` it would return `true`. The program does not use the return value; both cases were verified by printing them in a separate experiment.

Thanks to `IF NOT EXISTS`, the program raises no error when it is run a second time. Without this clause, the second run would give this message: `[SQLITE_ERROR] SQL error or missing database (table users already exists)`.

11.6.3 Inserting data

```
String url = "jdbc:sqlite:sample.db";
String sql = "INSERT INTO users(name, email) "
           + "VALUES('Ahmet Yılmaz', 'ahmet@example.com')";
try (Connection conn = DriverManager.getConnection(url);
     Statement stmt = conn.createStatement()) {
  // Insert data into the database.
  int n = stmt.executeUpdate(sql);
  System.out.println("Data inserted! (" + n + " row)");
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

```
Data inserted! (1 row)
```

The insert is done with `executeUpdate`. This method is used for `INSERT`, `UPDATE`, and `DELETE` and returns the number of rows the statement affected, as an `int`. Since a single row was inserted, the returned value is 1. No `id` was given; SQLite assigns 1 because the table is empty.

The SQL text is written in Java as two pieces joined with `+`. Note the space at the end of the first piece: `"INSERT INTO users(name, email) "`. If the space were forgotten, the text would join as `…email)VALUES(…`. SQLite would accept that, but elsewhere, for example in `"FROM users" + "WHERE id = 1"`, `usersWHERE` appears and the query breaks. Text values are in single quotes, as SQL requires.

Each time the program runs, a new row is inserted: `id` 2, 3, … The same person ends up in the table many times. This can change query results in unexpected ways.

11.6.4 Reading data

```
String url = "jdbc:sqlite:sample.db";
String sql = "SELECT * FROM users";
try (Connection conn = DriverManager.getConnection(url);
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery(sql)) {
  // Read and print the rows one by one.
  while (rs.next()) {
    System.out.println("ID: " + rs.getInt("id")
        + ", Name: " + rs.getString("name")
        + ", Email: " + rs.getString("email"));
  }
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

```
ID: 1, Name: Ahmet Yılmaz, Email: ahmet@example.com
```

Reading is done with `executeQuery`, which returns a `ResultSet`. This time the result set is also a resource, so it joins the try-with-resources parentheses as the third resource. `rs.next()` moves the cursor to the next row; it returns `true` if there is a row and `false` if not. `getInt("id")` and `getString("name")` read the columns of the current row by name. Since the table has a single row, the loop runs once.

Reading by column name is safer than reading by position. `SELECT *` returns the columns in the order of the table definition; if a new column is added to the table later, the numbers may shift, while the names do not change. In the SQLite driver, column names are case-insensitive: `getString("NAME")` works too. A column name that does not exist throws a `SQLException`: trying `getString("title")` gave the message `no such column: 'title'`.

11.6.5 Code trace: reading with `ResultSet`

The `while (rs.next())` loop is the most common JDBC pattern. The following program reads a table of two rows. `reading.db` was prepared before the program with two rows: (1, Ahmet Yılmaz) and (2, Ayşe Demir).

```
String url = "jdbc:sqlite:reading.db";
String sql = "SELECT * FROM users";
try (Connection conn = DriverManager.getConnection(url);
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery(sql)) {
  while (rs.next()) {
    int id = rs.getInt("id");
    String name = rs.getString("name");
    System.out.println(id + " - " + name);
  }
  System.out.println("Done reading.");
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

```
1 - Ahmet Yılmaz
2 - Ayşe Demir
Done reading.
```

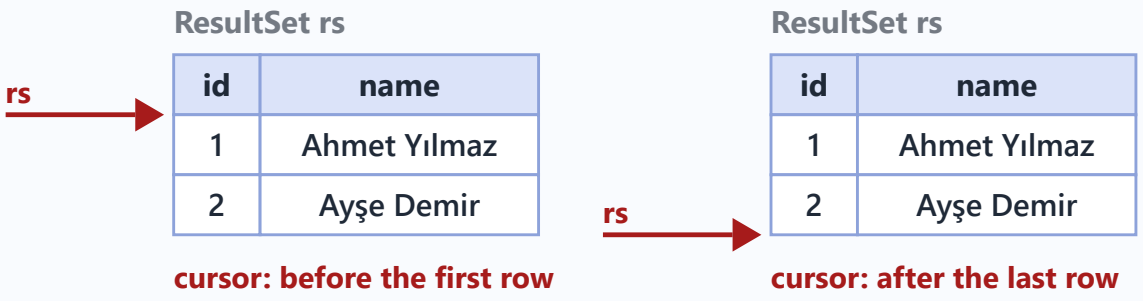


Figure 11.20: The cursor starts before the first row (left) and, after the last `next()` call, sits past the last row (right).

Step	Line	Variables	New output
1	3: `try (Connection conn = DriverManager.getConnection(url);` 4: `Statement stmt = conn.createStatement();` 5: `ResultSet rs = stmt.executeQuery(sql)) {`	—	—
2	6: `while (rs.next()) {`	rs.next()=true	—
3	7: `int id = rs.getInt("id");` 8: `String name = rs.getString("name");`	rs.next()=true, id=1, name="Ahmet Yılmaz"	—
4	9: `System.out.println(id + " - " + name);`	rs.next()=true, id=1, name="Ahmet Yılmaz"	`1 - Ahmet Yılmaz`
5	6: `while (rs.next()) {`	rs.next()=true, id=1, name="Ahmet Yılmaz"	—
6	7: `int id = rs.getInt("id");` 8: `String name = rs.getString("name");`	rs.next()=true, id=2, name="Ayşe Demir"	—
7	9: `System.out.println(id + " - " + name);`	rs.next()=true, id=2, name="Ayşe Demir"	`2 - Ayşe Demir`
8	6: `while (rs.next()) {`	rs.next()=false	—
9	11: `System.out.println("Done reading.");`	rs.next()=false	`Done reading.`
10	12: `} catch (SQLException e) {`	—	—

Table 11.15: Tracing the read with `ResultSet`.

Table 11.15 gives ten steps. In the first step the connection is opened and the query runs; `rs` holds the two-row result, and the cursor is before the first row (Figure 11.20, left). In the second step `next()` moves the cursor to the first row and returns `true`. In the third and fourth steps the columns are read from the current row and printed. In the fifth step the cursor moves to the second row; the sixth and seventh steps do the same work for the second row. In the eighth step there are no more rows; `next()` returns `false` and the loop ends. In the ninth step "Done reading." is printed. In the tenth step the `try` block ends and `rs`, `stmt`, and `conn` are closed in that order.

`next()` was called three times in total: once for each of the two rows and once to find out that the rows had ended. In general, for a result of *n* rows the loop runs *n* times and `next()` is called *n* + 1 times. For an empty result, the first `next()` returns `false` and the loop body never runs. This is not an error: a query that matches nothing, such as `WHERE id = 99`, returns an empty result set.

> **Common mistake**
>
> Reading a column without calling `next()` first. The cursor is before the first row; there is no row to read. Standard JDBC expects a `SQLException` in this case. The SQLite driver 3.40.0.0, however, returned the first row's value without an error in the test. This is one driver's own behavior, and on other drivers the code fails. For a single-row result, too, the pattern is `if (rs.next()) { … }`.

11.6.6 Updating and deleting data

Updating and deleting use the same pattern as inserting; only the SQL text changes.

```
String url = "jdbc:sqlite:sample.db";
String sql = "UPDATE users SET name = 'Mehmet Akdeniz' WHERE id = 1";
try (Connection conn = DriverManager.getConnection(url);
     Statement stmt = conn.createStatement()) {
  // Update the data in the database.
  int n = stmt.executeUpdate(sql);
  System.out.println("Data updated! (" + n + " row)");
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

```
Data updated! (1 row)
```

Before the update, the read program printed `ID: 1, Name: Ahmet Yılmaz, …`. After the update, the same program prints this:

```
ID: 1, Name: Mehmet Akdeniz, Email: ahmet@example.com
```

`SET` changes only the column named; `email` stays the same. `WHERE id = 1` selects the row to update. `executeUpdate` returns 1 because a single row satisfied the condition. The delete program has the same structure:

```
String url = "jdbc:sqlite:sample.db";
String sql = "DELETE FROM users WHERE id = 1";
try (Connection conn = DriverManager.getConnection(url);
     Statement stmt = conn.createStatement()) {
  // Delete the data from the database.
  int n = stmt.executeUpdate(sql);
  System.out.println("Data deleted! (" + n + " row)");
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

When the program is run twice, the output is as follows:

```
Data deleted! (1 row)
```

```
Data deleted! (0 row)
```

On the first run, the row with number 1 is deleted. On the second run, no row is left to delete. This is not an error: `DELETE` deletes the zero rows that satisfy the condition, and `executeUpdate` returns 0. The program still prints "Data deleted!", which misleads the user. The message should therefore depend on the returned count. A `DELETE FROM users` without `WHERE` deletes all rows in the table. The table itself, its structure, remains; the statement that removes a table entirely is `DROP TABLE`.

11.6.7 The three `execute` methods

`Statement` has three methods that run SQL. Table 11.16 shows when to use which.

Method	Returns	Use for
`execute`	`boolean`: is there a result set	`CREATE TABLE`, `DROP`; SQL of unknown kind
`executeUpdate`	`int`: the number of rows affected	`INSERT`, `UPDATE`, `DELETE`
`executeQuery`	`ResultSet`	`SELECT`

Table 11.16: The three execution methods of `Statement`.

What happens if the wrong method is chosen? The following experiment calls each method with a "wrong" statement:

```
query does not return ResultSet
row count: 1
executeUpdate(SELECT) = 0
execute(INSERT) = false, getUpdateCount = 1
execute(SELECT) = true
execute(CREATE) = false, getUpdateCount = 1
executeUpdate(CREATE) = 0
executeUpdate(DROP) = 0
```

Running `INSERT` with `executeQuery` throws a `SQLException` with the message `query does not return ResultSet`. The second line is surprising: despite the error, the row was inserted into the table. The SQLite driver ran the statement first and noticed only afterward that there was no result set. It would be wrong for a program to assume "an error occurred, so nothing was inserted". `executeUpdate` with `SELECT` raised no error with this driver and returned 0; the standard expects a `SQLException`. `execute` accepts every statement and lets you obtain the result afterward with `getUpdateCount()` or `getResultSet()`. For `CREATE` and `DROP`, `executeUpdate` returns 0. In short, choose the right method; drivers react to misuse in different ways.

When the six programs are run in sequence from the command line, the screen looks like this:

```
> java Connect
Connected!
> java CreateTable
Table created!
> java InsertData
Data inserted! (1 row)
> java ReadData
ID: 1, Name: Ahmet Yılmaz, Email: ahmet@example.com
> java UpdateData
Data updated! (1 row)
> java ReadData
ID: 1, Name: Mehmet Akdeniz, Email: ahmet@example.com
> java DeleteData
Data deleted! (1 row)
> java ReadData
> java DeleteData
Data deleted! (0 row)
```

This transcript is identical to the real run. After the delete, `ReadData` prints nothing: the table exists but is empty, and the loop body never runs. The last `DeleteData` deletes 0 rows.

11.6.8 Code trace: return values of `executeUpdate`

The number that `executeUpdate` returns carries the information the program needs to give the right message. The following code inserts three records into a table of supplies, then performs an update and a delete. The insert uses a `PreparedStatement`. All you need to know here is that the `?` marks

stand in for values and are filled with `setString` and `setInt`.

```
String[] names = {"stapler", "scissors", "tape"};
int[] stock = {40, 0, 15};
PreparedStatement ins = conn.prepareStatement(
    "INSERT INTO supplies(name, stock) VALUES(?, ?)");
for (int i = 0; i < names.length; i++) {
  ins.setString(1, names[i]);
  ins.setInt(2, stock[i]);
  ins.executeUpdate();
}
int updated = stmt.executeUpdate(
    "UPDATE supplies SET stock = stock + 10 WHERE stock < 20");
int deleted = stmt.executeUpdate("DELETE FROM supplies WHERE stock > 100");
System.out.println(updated + " updated, " + deleted + " deleted");
```

When the program was run with a connection and a `supplies(id, name, stock)` table, the value that `executeUpdate` returned in each round of the insert loop was printed as well. The program's last line and the final state of the table are in the output too:

```
round 0: 1
round 1: 1
round 2: 1
2 updated, 0 deleted
1 stapler 40
2 scissors 10
3 tape 25
```

Step	Line	Variables	New output
1	1: `String[] names = {"stapler", "scissors", "tape"};` 2: `int[] stock = {40, 0, 15};` 3: `PreparedStatement ins = conn.prepareStatement(` 4: `"INSERT INTO supplies(name, stock) VALUES(?, ?)");`	names=[stapler, scissors, tape], stock=[40, 0, 15]	—
2	6: `ins.setString(1, names[i]);` 7: `ins.setInt(2, stock[i]);`	i=0	—
3	8: `ins.executeUpdate();`	i=0, returned=1	—
4	5: `for (int i = 0; i < names.length; i++) {` 6: `ins.setString(1, names[i]);` 7: `ins.setInt(2, stock[i]);` 8: `ins.executeUpdate();`	i=1, returned=1	—
5	5: `for (int i = 0; i < names.length; i++) {` 6: `ins.setString(1, names[i]);` 7: `ins.setInt(2, stock[i]);` 8: `ins.executeUpdate();`	i=2, returned=1	—

Step	Line	Variables	New output
6	10: `int updated = stmt.executeUpdate(` 11: `"UPDATE supplies SET stock = stock + 10 WHERE stock < 20");`	updated=2	—
7	12: `int deleted = stmt.executeUpdate("DELETE FROM supplies WHERE stock > 100");`	updated=2, deleted=0	—
8	13: `System.out.println(updated + " updated, " + deleted + " deleted");`	updated=2, deleted=0	`2 updated, 0 deleted`

Table 11.17: Tracing the return values of `executeUpdate`.

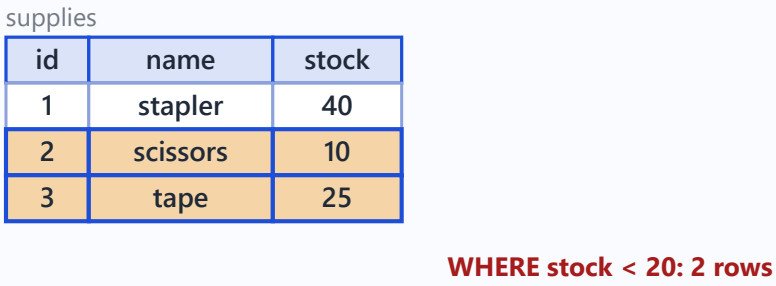


Figure 11.21: `WHERE stock < 20` selects the scissors (0) and tape (15) rows; 10 is added to both.

The insert statement is prepared once before the loop; in each round only the values change. Each insert returns 1, and SQLite gives the rows the numbers 1, 2, and 3 in order. In the update, the condition `WHERE stock < 20` is satisfied by scissors (0) and tape (15) (Figure 11.21). Ten is added to the stock of both: scissors becomes 10 and tape becomes 25. `updated` is 2. `SET stock = stock + 10` computes the new value from the column's old value; it is the counterpart of `stock += 10` in Java.

In the delete, no product has a stock above 100. No error occurs; 0 is simply returned. By looking at these counts, the program can give correct messages such as "no record found". A note on cost: if the `WHERE` condition is on a column with no index (such as `stock`), update and delete scan the whole table; in a table of *n* rows the cost is proportional to *n*. With a primary key condition such as `WHERE id = 1`, the row is found directly.

11.7 `PreparedStatement`

The SQL texts so far were fixed. In real programs, the values come from the user: the name to search for, the grade to insert, the number of the record to delete. There is a wrong way and a right way to put these values into the SQL text.

11.7.1 SQL injection

The easiest-looking way is to paste the value the user typed into the SQL text by string concatenation. On a search screen the user types the name to look for, and the program builds this text: `"SELECT name FROM people WHERE name = '" + input + "'"`. As long as the input is `Ali`, all is well. But if the input contains a quotation mark, the structure of the SQL text changes.

> **Definition 11.10**
>
> **SQL injection** is an attack in which user input gets mixed into an SQL statement and changes its meaning. The input is interpreted not as a value but as part of the SQL.

```
String input = "x' OR '1'='1";   // the "name" the user typed

// DANGEROUS: the input is pasted into the SQL text
String sql = "SELECT name FROM people WHERE name = '"
           + input + "'";
System.out.println(sql);
ResultSet rs1 = stmt.executeQuery(sql);

// SAFE: the input is sent only as a value
PreparedStatement ps = conn.prepareStatement(
    "SELECT name FROM people WHERE name = ?");
ps.setString(1, input);
ResultSet rs2 = ps.executeQuery();
```

The first half of the code is the dangerous way and the second half is the safe way. With three people named Ali, Ayşe, and Can in the `people` table, the rows of the two result sets were counted:

```
SELECT name FROM people WHERE name = 'x' OR '1'='1'
Concatenation: 3 rows
PreparedStatement: 0 rows
input length: 12
[SQLITE_ERROR] SQL error or missing database (near "s": syntax error)
with PreparedStatement: 1
Ali's
INSERT INTO people(name) VALUES('Ali''s')
table still exists: 4
```

The first line of the output is the SQL text that went to the database. The user typed `x' OR '1'='1` in place of a name. In the concatenation, the first quotation mark in the input closes the quotation mark the program opened. The name value ends as `'x'`. The `OR '1'='1'` that follows is no longer a value

but part of the SQL. The quotation mark the program appended at the very end closes `'1`, and the text becomes syntactically perfect. The condition is `name = 'x' OR '1'='1'`; because `'1'='1'` is true in every row, the condition eliminates no row. Three rows come back.

On a login screen this means signing in without a password. Worse is possible too: the input could be a piece of SQL that deletes data or reads passwords from other tables. SQL injection has for years been among the most common security holes in web applications.

In the second half, the `PreparedStatement` returns 0 rows with the same input. Here the SQL text is fixed and the place of the input is marked with `?`. The input is sent *separately* with `setString(1, input)` and used only as a value. The database looks for "the person whose name is exactly `x' OR '1'='1`". This 12-character text is nobody's name.

The last lines of the output show two more side results. When a text with an apostrophe, such as `Ali's`, is inserted by concatenation, the SQL breaks and the message `near "s": syntax error` is received. Even an ordinary user with no intent to attack can break the program. With `PreparedStatement` the same text is inserted and read back without trouble. Doubling the quote by hand (`''`) is another way, but it is error-prone; a single forgotten spot leaves the whole program vulnerable. The SQLite driver's `executeQuery` did not run a second statement appended with a semicolon (after the `DROP TABLE` attempt the table was still there). This does not count as protection: an attack that stays inside a single statement, like the one above, still works.

11.7.2 Code trace: SQL injection

The following code compares the two ways by row count:

```
String input = "x' OR '1'='1";
String sql = "SELECT name FROM people WHERE name = '"
           + input + "'";
int n = 0;
try (ResultSet rs = stmt.executeQuery(sql)) {
  while (rs.next()) n++;
}
System.out.println("Concatenation: " + n + " rows");
PreparedStatement ps = conn.prepareStatement(
    "SELECT name FROM people WHERE name = ?");
ps.setString(1, input);
n = 0;
try (ResultSet rs = ps.executeQuery()) { while (rs.next()) n++; }
System.out.println("PreparedStatement: " + n + " rows");
```

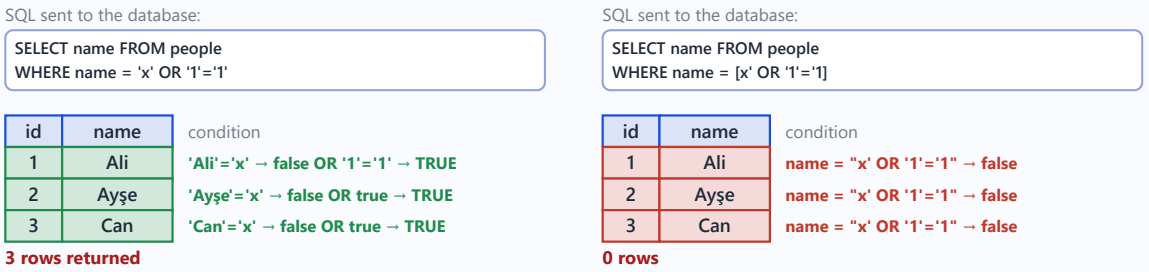


Figure 11.22: In the concatenated SQL the condition is true in every row and 3 rows come back (left); in the `PreparedStatement` the input is a single text value and matches no name (right).

Step	Line	Variables	New output
1	1: `String input = "x' OR '1'='1";`	input=x' OR '1'='1	—
2	2: `String sql = "SELECT name FROM people WHERE name = '"` 3: `+ input + "'";`	sql=SELECT … WHERE name = 'x' OR '1'='1'	—
3	5: `try (ResultSet rs = stmt.executeQuery(sql)) {`	rs=ResultSet	—
4	5: `try (ResultSet rs = stmt.executeQuery(sql)) {`	rs=ResultSet	—
5	6: `while (rs.next()) n++;` 7: `}`	n=3	—
6	8: `System.out.println("Concatenation: " + n + " rows");`	n=3	`Concatenation: 3 rows`
7	9: `PreparedStatement ps = conn.prepareStatement(` 10: `"SELECT name FROM people WHERE name = ?");`	ps=PreparedStatement	—
8	11: `ps.setString(1, input);`	1. ?=x' OR '1'='1	—
9	13: `try (ResultSet rs = ps.executeQuery()) { while (rs.next()) n++; }`	n=0	—
10	14: `System.out.println("PreparedStatement: " + n + " rows");`	n=0	`PreparedStatement: 0 rows`

Table 11.18: Tracing SQL injection.

In the third and fourth steps, the database evaluates the condition for each row. For Ali, `'Ali'='x'` is false, but the right side of the `OR` is always true; the row is selected. Ayşe and Can are selected for the same reason (Figure 11.22, left). In the seventh step the SQL is compiled *first*; `?` is only the place of a value. The structure of the statement is settled at this point and cannot change, whatever the input. In the eighth step the input is bound as a single text value as a whole; the quotation marks in it are ordinary characters. In the ninth step no name equals this text. The same input gives two different results. The rule is firm: a value that comes from the user is always given to SQL with `?`.

11.7.3 Placeholders

Definition 11.11

A **prepared statement** (`PreparedStatement`) is an SQL statement that contains `?` placeholders in place of values and takes the values separately. It is created with the connection's `prepareStatement(sql)` method; the placeholders are filled with `setXxx(index, value)` methods.

The following method inserts a person into the `people` table:

```
static void add(Connection conn, String name, String surname,
                int age) throws SQLException {
  String sql = "INSERT INTO people(name, surname, age) "
             + "VALUES(?, ?, ?)";
  try (PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setString(1, name);      // 1st question mark
    ps.setString(2, surname);   // 2nd question mark
    ps.setInt(3, age);          // 3rd question mark
    ps.executeUpdate();
  }
}
```

There are three `?` in the SQL text. The first argument of the `setXxx` methods is the index of the placeholder; the index starts **at 1** and follows the left-to-right order in the SQL text (Figure 11.23). The method name says the Java type of the value: `setString` for text, `setInt` for an integer, `setDouble` for a floating-point number. After all placeholders are filled, `executeUpdate()` is called *without parameters*, because the SQL text was given when the statement was prepared.

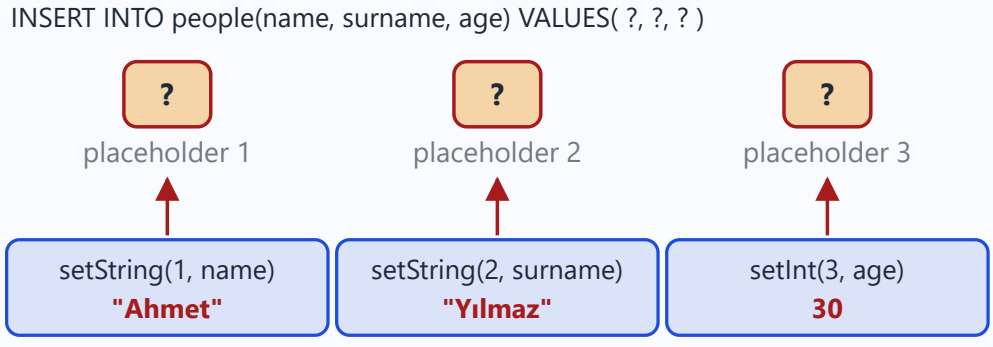


Figure 11.23: The `setString` and `setInt` calls fill the question marks in order, with numbers starting at 1.

`PreparedStatement` has three benefits. The first is **security**: it prevents SQL injection. The second is that there is **no quoting problem**: `"Ali's"` is saved without trouble. The third is **reuse**: the same statement is prepared once and run again and again with different values. The DBMS does not parse the statement anew each time; the difference is noticeable when thousands of inserts are made in a loop.

Placeholders can stand in only for *values*. A table or column name cannot be given with `?`: preparing `SELECT * FROM ?` gave the message `near "?": syntax error`. A `'?'` inside quotes is not a placeholder but a question-mark character; in the query `WHERE name = '?'`, a `setString(1, …)` call fails. A placeholder that is not filled counts as `NULL`. When the insert with three placeholders was run with only two values given, the message `NOT NULL constraint failed: people.age` was received. One more detail: a value that has been set stays in place until a new `setXxx` call. When, after an insert, only `setString(1, "Bora")` was changed and the statement was run again, the second row took the previous row's surname and age.

Common mistake

Starting the placeholder index at 0: `ps.setInt(0, 5)`. The standard expects a `SQLException` in this case. The SQLite driver 3.40.0.0, however, threw an unchecked `ArrayIndexOutOfBoundsException` (`Index -1 out of bounds for length 3`). This exception is not caught by `catch (SQLException e)` and stops the program. The index always starts at 1 and does not exceed the number of `?`.

11.7.4 The people program

The insert method, together with methods for updating, deleting, and listing, makes up a small but complete program. The update and delete methods are as follows:

```
static void update(Connection conn, String name, String surname,
                   int age) throws SQLException {
  String sql = "UPDATE people SET surname = ?, age = ? "
             + "WHERE name = ?";
  try (PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setString(1, surname);
    ps.setInt(2, age);
    ps.setString(3, name);
    ps.executeUpdate();
  }
}

static void delete(Connection conn, String name) throws SQLException {
  String sql = "DELETE FROM people WHERE name = ?";
  try (PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setString(1, name);
    ps.executeUpdate();
  }
}
```

Each method takes the `Connection` as a parameter. The connection is opened once in `main` and used for many jobs. Opening a new connection for every operation is both slow and unnecessary. A `PreparedStatement`, on the other hand, is opened in each method's own try-with-resources block and closed when the method ends; the connection stays open.

In the update method, pay attention to the order of the `?` marks. The SQL text is `SET surname = ?, age = ? WHERE name = ?`. Although the method's parameter order is `(name, surname, age)`, in the SQL the name is the *third* placeholder. That is why `setString(3, name)` is written. The numbers follow the SQL text, not the Java parameters.

The methods pass errors to the caller with `throws SQLException`. Because `main` also declares `throws SQLException`, if an error occurs the program stops with a stack trace. For a small example program this is acceptable. In a real program the error is caught in `main` and the user is given an understandable message. The listing method is as follows:

```
static void list(Connection conn) throws SQLException {
  String sql = "SELECT id, name, surname, age FROM people";
  try (Statement st = conn.createStatement();
       ResultSet rs = st.executeQuery(sql)) {
    while (rs.next()) {
      System.out.println(rs.getInt("id") + " " + rs.getString("name")
          + " " + rs.getString("surname") + " " + rs.getInt("age"));
    }
    System.out.println("---");
  }
}
```

This query has no variable value; a plain `Statement` is enough. Since there is no user input, there is no injection risk either. Each call to `list` reads the table as it is at that moment and prints `---` at the end. Finally, `main` calls the methods in order:

```
public static void main(String[] args) throws SQLException {
  String url = "jdbc:sqlite:people.db";
  try (Connection conn = DriverManager.getConnection(url);
       Statement stmt = conn.createStatement()) {
    stmt.execute("CREATE TABLE IF NOT EXISTS people ("
        + "id INTEGER PRIMARY KEY, name TEXT NOT NULL, "
        + "surname TEXT NOT NULL, age INTEGER NOT NULL)");
    add(conn, "Ahmet", "Yılmaz", 30);
    add(conn, "Ayşe", "Demir", 40);
    update(conn, "Ahmet", "Akdeniz", 35);
    list(conn);
    delete(conn, "Ahmet");
    list(conn);
  }
}
```

The four methods and `main` were combined in a single `Person` class and run in an empty folder:

```
1 Ahmet Akdeniz 35
2 Ayşe Demir 40
---
2 Ayşe Demir 40
---
```

11.7.5 Code trace: the people program

Step	Line	Variables	New output
1	3: `try (Connection conn = DriverManager.getConnection(url);` 4: `Statement stmt = conn.createStatement()) {`	—	—
2	5: `stmt.execute("CREATE TABLE IF NOT EXISTS people ("` 6: `+ "id INTEGER PRIMARY KEY, name TEXT NOT NULL, "` 7: `+ "surname TEXT NOT NULL, age INTEGER NOT NULL)");`	—	—
3	8: `add(conn, "Ahmet", "Yılmaz", 30);`	—	—
4	9: `add(conn, "Ayşe", "Demir", 40);`	—	—
5	10: `update(conn, "Ahmet", "Akdeniz", 35);`	—	—

Step	Line	Variables	New output
6	11: `list(conn);`	—	`1 Ahmet Akdeniz 35 ⏎ 2 Ayşe Demir 40 ⏎ ---`
7	12: `delete(conn, "Ahmet");`	—	—
8	13: `list(conn);`	—	`2 Ayşe Demir 40 ⏎ ---`
9	14: `}`	—	—

Table 11.19: Tracing the people program (line numbers refer to `main`).

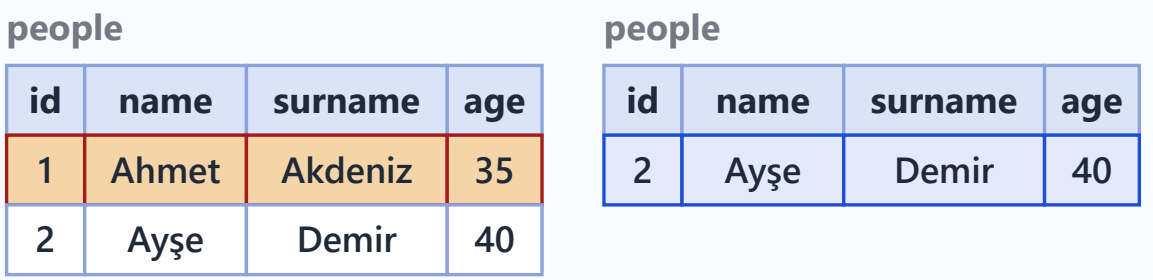


Figure 11.24: The `people` table after the update (left) and after the delete (right).

In the first step, `people.db` is created if it does not exist; the connection and `stmt` are opened. In the second step the table is created and is still empty. In the third and fourth steps `add` is called; Ahmet is inserted with `id` = 1 and Ayşe with `id` = 2. In the fifth step `WHERE name = ?` selects Ahmet; his surname becomes Akdeniz and his age 35. In the sixth step `list` prints the two rows and `---`. In the seventh step Ahmet's row is deleted. In the eighth step only Ayşe is left in the table. In the ninth step the connection is closed; the data persists in `people.db` (Figure 11.24).

A note on cost can be added. `update` and `delete` work with `WHERE name = ?`. Because there is no index on the `name` column, each call scans the table. Moreover, if two people have the same name, both are updated or deleted. Working with the primary key (`WHERE id = ?`) is both fast and exact.

11.8 Advanced JDBC

11.8.1 Generated keys: `getGeneratedKeys`

The database decides the `id` of a new row. A program often wants to know this number right away. When an order is inserted, its line items go into a separate table together with the order's `id`. The way to learn the number is `getGeneratedKeys()` :

```
String sql = "INSERT INTO students(name, dept_id) "
    + "VALUES(?, ?)";
try (PreparedStatement ps = conn.prepareStatement(sql,
        Statement.RETURN_GENERATED_KEYS)) {
  ps.setString(1, "Ece");
  ps.setInt(2, 3);
  ps.executeUpdate();
  try (ResultSet k = ps.getGeneratedKeys()) {
    if (k.next()) {
      System.out.println("New id: " + k.getInt(1));
    }
  }
}
```

```
New id: 5
id INTEGER PRIMARY KEY -> 3
id INTEGER PRIMARY KEY AUTOINCREMENT -> 4
after deleting from the middle: 1,3,4
last_insert_rowid: 5
```

`Statement.RETURN_GENERATED_KEYS` is passed to `prepareStatement` as the second argument. This constant tells the driver to keep the generated keys. After `executeUpdate`, `getGeneratedKeys()` returns a `ResultSet`. The single row and single column of this result set hold the new key. The table already had four students (Ada, Bora, Cem, Deniz); Ece got number 5. Because the new key is also a result set, it is opened and closed with try-with-resources.

The next two lines of the output show a subtlety of numbering. Rows 1, 2, and 3 are inserted into a table, 3 is deleted, and a new row is inserted. Which number does the new row get? Table 11.20 compares the two definitions.

Column definition	New `id`
`id INTEGER PRIMARY KEY`	3 (largest `id` + 1)
`id INTEGER PRIMARY KEY AUTOINCREMENT`	4 (a used number is never given again)

Table 11.20: 1, 2, 3 inserted, 3 deleted; the `id` of the new row.

Plain `INTEGER PRIMARY KEY` gives one more than the largest existing number. If the largest number is deleted, that number is reused. If a number in the middle is deleted (for example 2), the new row still gets 4; the gap is not filled. `AUTOINCREMENT` stores the largest number used so far in a separate table and never hands out a number again. If the number of a deleted record must not pass to another record (an invoice number, an order number), `AUTOINCREMENT` is used. Its counterpart in MySQL is `AUTO_INCREMENT`.

11.8.2 From rows to objects: the DAO pattern

The programs so far scattered SQL and row reading throughout `main`. In an object-oriented program, each row is instead converted to an object, and the SQL is collected in one place. A simple class is enough for student rows:

```
class Student {
  private final int id;
  private final String name;
  Student(int id, String name) {
    this.id = id;
    this.name = name;
  }
  public String toString() {
    return "#" + id + " " + name;
  }
}
```

The fields of the `Student` class are `final`; the object does not change after it is created. `toString` writes the object in the form `#1 Ada`. This class knows nothing about the database: it contains neither SQL nor JDBC.

Definition 11.12

A **data access object** (DAO) is a design that collects all of a table's SQL operations in a single class and returns the results as objects. The rest of the program sees no SQL; it only calls the DAO's methods.

Table 11.21 lists the methods of a `StudentDao` class. Each method corresponds to one SQL statement and returns the result in Java's terms: an object instead of a row, a list instead of a result set, a `boolean` instead of the number of rows affected.

`StudentDao` method	SQL	Returns
`add(name, dept)`	`INSERT`	the new `id`
`find(id)`	`SELECT … WHERE id = ?`	`Student` or `null`
`inDept(dept)`	`SELECT … WHERE dept_id = ?`	`List<Student>`
`delete(id)`	`DELETE`	whether a row was deleted (`boolean`)

Table 11.21: The methods of the `StudentDao` class.

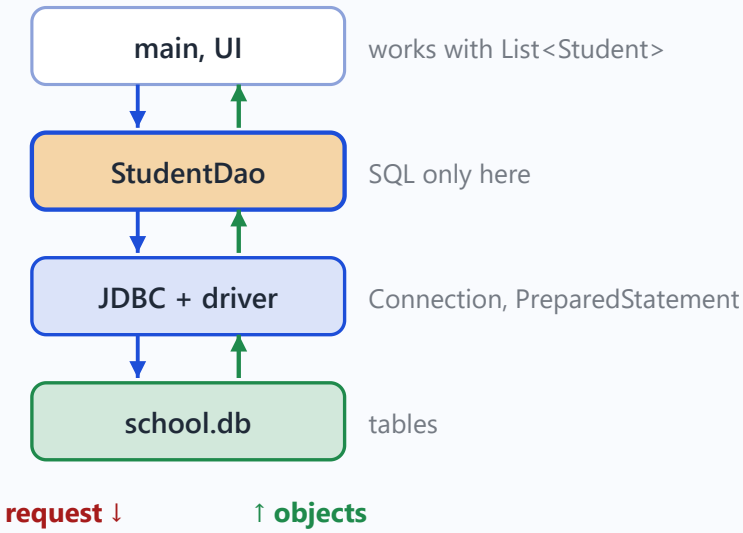


Figure 11.25: The layers: user interface, `StudentDao`, JDBC, and the database file. Requests go down, objects go up.

Figure 11.25 shows the layered structure. The user interface (`main` or a Swing window) knows only `StudentDao` and asks it for a `List<Student>`. The DAO works with JDBC and SQL. The SQL is only in this class. If the database changes or the tables are reorganized, only the DAO changes; the interface code stays the same. This is the database-level counterpart of encapsulation. The connection is given to the DAO through its constructor; the DAO neither opens nor closes the connection.

All four methods in the table were written and run. On a table holding Ada, Bora, Cem, and Deniz, the results are as follows:

```
add -> 5
find(2) -> #2 Bora
find(9) -> null
inDept(1) -> [#1 Ada, #3 Cem, #5 Ece]
delete(3) -> true
delete(3) -> false
inDept(1) -> [#1 Ada, #5 Ece]
```

`add` returns Ece's number (5) through `getGeneratedKeys`. `find(9)` finds no row and returns `null`; the calling code must check for it. `delete(3)` returns `true` on the first call and `false` on the second, because `executeUpdate` gives 0 the second time. The DAO converts this count to a `boolean`, shielding the rest of the program from the details of SQL.

In this example, mapping rows to objects was done by hand. In large projects, **ORM** (object-relational mapping) libraries do this automatically, for example the JPA standard and Hibernate, one of its implementations. Annotations declare which columns the class fields correspond to, and the library generates most of the SQL itself. JDBC still runs behind an ORM; what you learn in this chapter is also the foundation for understanding those libraries.

11.8.3 Code trace: from row to object

The following method returns the students in a department as a list of objects; the last two lines call it from `main`:

```
static List<Student> inDept(Connection conn,
    int dept) throws SQLException {
  List<Student> list = new ArrayList<>();
  String sql = "SELECT id, name FROM students "
      + "WHERE dept_id = ? ORDER BY id";
  try (PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setInt(1, dept);
    try (ResultSet rs = ps.executeQuery()) {
      while (rs.next()) {
        list.add(new Student(rs.getInt("id"),
                             rs.getString("name")));
      }
    }
  }
  return list;
}
// inside main:
List<Student> list = inDept(conn, 1);
for (Student s : list) System.out.println(s);
```

```
#1 Ada
#3 Cem
dept 3: []
dept 2: [#2 Bora]
```

Step	Line	Call stack	Variables	New output
1	18: `List<Student> list = inDept(conn, 1);`	main() → inDept(conn, 1)	dept=1	—
2	3: `List<Student> list = new ArrayList<>();` 4: `String sql = "SELECT id, name FROM students "` 5: `+ "WHERE dept_id = ? ORDER BY id";` 6: `try (PreparedStatement ps = conn.prepareStatement(sql)) {`	main() → inDept(conn, 1)	dept=1, list=[]	—
3	7: `ps.setInt(1, dept);` 8: `try (ResultSet rs = ps.executeQuery()) {`	main() → inDept(conn, 1)	dept=1, list=[]	—
4	9: `while (rs.next()) {` 10: `list.add(new Student(rs.getInt("id"),` 11: `rs.getString("name")));`	main() → inDept(conn, 1)	list=[#1 Ada]	—
5	9: `while (rs.next()) {` 10: `list.add(new Student(rs.getInt("id"),` 11: `rs.getString("name")));`	main() → inDept(conn, 1)	list=[#1 Ada, #3 Cem]	—

Step	Line	Call stack	Variables	New output
6	9: `while (rs.next()) {` 13: `}` 14: `}`	main() → inDept(conn, 1)	list=[#1 Ada, #3 Cem]	—
7	15: `return list;` 18: `List<Student> list = inDept(conn, 1);`	main()	list=[#1 Ada, #3 Cem]	—
8	19: `for (Student s : list) System.out.println(s);`	main()	list=[#1 Ada, #3 Cem]	`#1 Ada ⏎ #3 Cem`

Table 11.22: Tracing the `inDept` method.

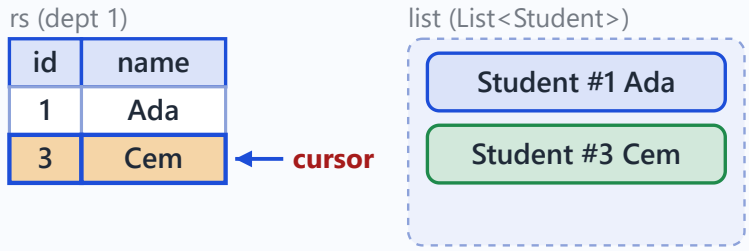


Figure 11.26: The second row (3, Cem) is read while the cursor is on it; the list holds two `Student` objects.

`main` calls the method with `dept` = 1; the connection is open in `main`. The method creates an empty list and prepares the query with `?`. `?` = 1 is bound and the query runs; department 1 has Ada and Cem. From the first row, `id` = 1 and `name` = Ada are read and `new Student(1, "Ada")` is added to the list. The second row produces the second object: `#3 Cem` (Figure 11.26). When `next()` returns `false`, the two nested `try` blocks end; first `rs` and then `ps` is closed. The connection is closed by `main`, which opened it. The method returns the two-element list, and its frame is removed from the stack. `main` prints each object with `toString()`; it knows nothing about SQL.

The last two lines of the output are the results of the same method for other departments. Department 3 (ME) has no students; the method returns an empty list, not `null`, and `[]` is printed. Returning an empty list is safer than `null`: if the calling code walks the list with a `for` loop, the loop simply does not run, and there is no risk of `NullPointerException`. The cost of the method is the cost of the query plus that of creating *k* objects, where *k* is the number of students in the department. The list takes memory for *k* objects. For very large results, it may be necessary to process rows one at a time instead of filling them all into a list.

11.8.4 The same code, another database: MySQL

The promise of JDBC is that the same code also works with other databases. The following program reads the `users` table on a MySQL server:

```
String url = "jdbc:mysql://localhost:3306/testdb";
String user = "root";
String password = "secret";
String sql = "SELECT id, name, email FROM users";
try (Connection conn =
         DriverManager.getConnection(url, user, password);
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery(sql)) {
  while (rs.next()) {
    System.out.println(rs.getInt("id") + " "
        + rs.getString("name"));
  }
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

The JDBC code is the same as in the SQLite example. Four things change: the connection URL, the user name, the password, and the driver jar on the classpath. MySQL's driver is `mysql-connector-j`. The URL has the form `jdbc:mysql://localhost:3306/testdb`: `localhost` is the machine the server runs on, 3306 is MySQL's default port, and `testdb` is the name of the database. The three-argument form of `getConnection` also takes the user name and password.

MySQL is a server. Before the program runs, the server must be installed and running, and the `testdb` database and the `users` table must have been created. There is no "create the file if it does not exist" behavior as in SQLite. The machine on which this chapter was prepared had no MySQL driver. When the program was compiled and run, it gave the message `No suitable driver found for jdbc:mysql://localhost:3306/testdb`; this message is the expected behavior.

The name of the driver class is `com.mysql.cj.jdbc.Driver`. The name `com.mysql.jdbc.Driver` seen in old code is deprecated. Since the driver has been loaded automatically since JDBC 4.0, this name is not written in the code anyway. One last warning: in real projects, the password is not written in the code. A password written in code is plainly visible in the compiled `.class` file and in the version control system. The password is read from a configuration file or an environment variable, for example with `System.getenv("DB_PASSWORD")`.

11.9 Transactions

So far, each SQL statement has run on its own and become permanent immediately. Some jobs, however, consist of several statements that make sense only together. This section explains how to run such jobs safely.

11.9.1 Transactions and ACID

Consider a money transfer: Ali sends 300 TL to Ece. The job is two `UPDATE` statements: 300 TL is added to Ece's account, and 300 TL is subtracted from Ali's. What happens if the power fails after the first statement has run, or if the second statement does not run because of an error? Ece's money has increased and Ali's has not decreased: money has appeared out of nothing. Had the order of the statements been reversed, money would have vanished. It is unacceptable for one of the two statements to happen and the other not.

Definition 11.13

A **transaction** is one or more SQL statements run as a single unit of work. Either all the statements of the transaction become permanent together, or none does. Making them permanent is called `commit`, and undoing them is called `rollback`.

In JDBC, the default setting of a connection is **auto-commit**: each statement is a transaction on its own and becomes permanent as soon as it runs. This setting was queried on a SQLite connection with `conn.getAutoCommit()` and came out `true`. The pattern for grouping several statements into one transaction is fixed. `conn.setAutoCommit(false)` starts a transaction, and then the statements run. If all goes well, `conn.commit()` makes the changes permanent; if an error occurs, `conn.rollback()` undoes them (Figure 11.27).

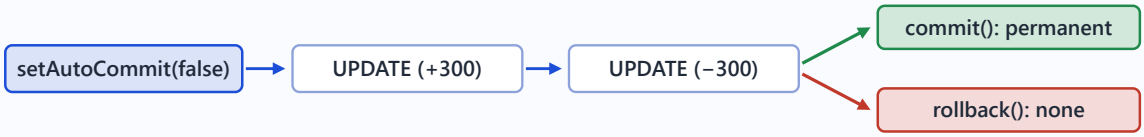


Figure 11.27: Transaction: the statements after `setAutoCommit(false)` are temporary; `commit()` makes them permanent and `rollback()` undoes them.

A reliable transaction has four properties, known by their initials as **ACID**.

- **Atomicity**: either the whole transaction is applied or none of it. Only one of the transfer's two statements cannot become permanent.
- **Consistency**: all constraints hold at the end of the transaction. If there is a `CHECK (balance >= 0)`, no transaction can end by pushing a balance below zero.
- **Isolation**: concurrent transactions do not see each other's half-done work. If another program reads Ece's balance while the transfer is under way, it sees either the value before the transfer or the value after it, never the one in between.
- **Durability**: committed data is not lost even if the power fails right afterward. The DBMS completes the commit only after the change has been safely written to disk.

Isolation can be seen in an experiment. Two separate connections were opened to the same `bank.db` file. The first connection turned off auto-commit and raised Ece's balance from 500 to 800, without committing yet. Meanwhile, both connections read the same row:

```
a sees: 800
b sees: 500
after commit b sees: 800
```

The connection that made the change sees its own temporary value (800). The second connection does not see the uncommitted change and reads 500. After the commit, the second connection also sees 800.

11.9.2 A money transfer

The transfer program has two parts. `main` sets up an in-memory database, inserts two accounts, and tries two transfers:

```
static final String DEPOSIT =
    "UPDATE accounts SET balance = balance + ? WHERE id = ?";
static final String WITHDRAW =
    "UPDATE accounts SET balance = balance - ? WHERE id = ?";

public static void main(String[] args) throws SQLException {
  String url = "jdbc:sqlite::memory:";  // in memory, temporary
  try (Connection conn = DriverManager.getConnection(url);
       Statement st = conn.createStatement()) {
    st.execute("CREATE TABLE accounts (id INTEGER PRIMARY KEY, "
        + "owner TEXT, balance INTEGER CHECK (balance >= 0))");
    st.execute("INSERT INTO accounts(owner, balance) "
        + "VALUES ('Ali', 1000), ('Ece', 500)");
    transfer(conn, 1, 2, 300);    // Ali → Ece 300 TL
    transfer(conn, 1, 2, 5000);   // Ali can't afford it
    printAll(conn);
  }
}
```

The `balance` column of the `accounts` table has the constraint `CHECK (balance >= 0)`. Any statement that pushes a balance below zero throws a `SQLException`. In the second transfer, Ali's 700 TL is not enough for a 5000 TL transfer. The `transfer` method is as follows:

```
static void transfer(Connection conn, int from, int to,
                     int amount) throws SQLException {
  conn.setAutoCommit(false);         // transaction starts
  try (PreparedStatement p1 = conn.prepareStatement(DEPOSIT);
       PreparedStatement p2 = conn.prepareStatement(WITHDRAW)) {
    p1.setInt(1, amount);  p1.setInt(2, to);
    p1.executeUpdate();              // deposit to receiver
    p2.setInt(1, amount);  p2.setInt(2, from);
    p2.executeUpdate();              // withdraw from sender
    conn.commit();                   // make permanent
    System.out.println(amount + " TL transferred.");
  } catch (SQLException e) {
    conn.rollback();                 // undo
    System.out.println("Rolled back: " + amount + " TL");
  } finally {
    conn.setAutoCommit(true);
  }
}
```

The method first turns off auto-commit; the transaction begins. With two `PreparedStatement`s it first deposits to the receiver and then withdraws from the sender. If both statements succeed, `commit()` makes both permanent together. If either statement throws a `SQLException`, control passes to `catch`, and `rollback()` undoes all the changes in the transaction. The `finally` block turns auto-commit back on no matter which path was taken; the connection returns to its default state for later users. The `printAll` method is a `SELECT` loop like `list` and prints each account's owner and balance. When run, the program gives this output:

```
300 TL transferred.
Rolled back: 5000 TL
Ali: 700
Ece: 800
```

11.9.3 Code trace: `commit` and `rollback`

Table 11.23 gives the balances and the state of the transaction through the two transfers. The "temporary" entries mark changes that have not been committed.

Step	Line	Event	Ali	Ece	State	New output
1	3	`setAutoCommit(false)`: first transfer, the transaction begins	1000	500	permanent	—
2	6–7	300 is deposited to Ece	1000	800	temporary	—
3	8–9	300 is withdrawn from Ali; the constraint holds	700	800	temporary	—

Step	Line	Event	Ali	Ece	State	New output
4	10–11	`commit()`: both changes become permanent together	700	800	permanent	`300 TL transferred.`
5	16	`finally`: auto-commit is turned back on	700	800	permanent	—
6	3, 6–7	Second transfer: 5000 is deposited to Ece	700	5800	temporary	—
7	8–9	700 − 5000 < 0: `CHECK` is violated, `SQLException`	700	5800	temporary	—
8	12–14	`rollback()`: the temporary deposit is undone	700	800	permanent	`Rolled back: 5000 TL`

Table 11.23: Tracing the transfer program (line numbers refer to the `transfer` method).

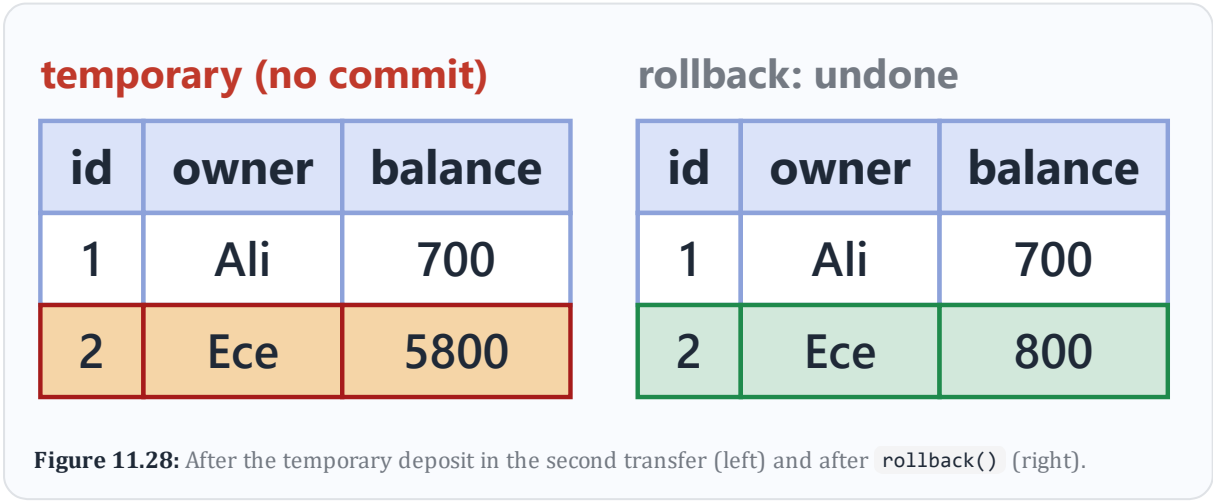


Figure 11.28: After the temporary deposit in the second transfer (left) and after `rollback()` (right).

The first transfer is trouble-free: the two statements are applied temporarily, and `commit()` makes both permanent together. In the second transfer, the deposit succeeds and Ece's balance rises temporarily to 5800 (Figure 11.28, left). The withdrawal would make Ali's balance −4300, so it runs into the `CHECK` constraint and a `SQLException` is thrown. Ali's balance never changes: the rejected statement is not applied. The `commit()` line is never reached. The `rollback()` in the `catch` block also undoes the deposit made so far in the transaction; Ece's balance returns to 800. Money has neither vanished nor multiplied: there was 1500 TL at the start, and at the end there is 700 + 800 = 1500 TL.

The value of this program becomes clear when you look at what would happen without a transaction. `setAutoCommit(false)`, `commit()`, and `rollback()` were removed from the same method and the program was run again:

```
300 TL transferred.
Rolled back: 5000 TL
  ([SQLITE_CONSTRAINT_CHECK] A CHECK constraint failed (CHECK constraint failed: balance
>= 0))
Ali: 700
Ece: 5800
```

In the second transfer, the deposit became permanent the moment it ran. The withdrawal was rejected, but there was no transaction to roll back. The program still printed "Rolled back", but wrongly: Ece's account holds 5800 TL, and 5000 TL has appeared out of nothing. The total money has risen to 6500 TL.

Common mistake

Forgetting `rollback()` in the `catch` block. If a statement fails while auto-commit is off and `rollback()` is not called, the transaction's successful statements wait in a pending state. Then the `setAutoCommit(true)` in `finally` runs. By JDBC rules, turning auto-commit on during a transaction *commits* it. In the test, the 5000 TL deposited to Ece became permanent this way: Ali 1000, Ece 5500. The error was caught, but the data was corrupted. Always call `rollback()` on the error path.

11.9.4 Batch processing

Writing 1000 values from a measuring device into a table takes 1000 `INSERT`s. These statements can be sent one by one, or collected in a queue and sent together.

Definition 11.14

A **batch update** collects many statements in a queue with `addBatch()` and sends them together with `executeBatch()`. `executeBatch()` returns the number of rows each statement affected, as an `int[]` array.

```
conn.setAutoCommit(false);
try (PreparedStatement ps = conn.prepareStatement(
        "INSERT INTO readings(value) VALUES(?)")) {
  for (int i = 1; i <= 1000; i++) {
    ps.setInt(1, i);
    ps.addBatch();
  }
  int[] result = ps.executeBatch();
  conn.commit();
  System.out.println(result.length + " statements, "
      + "the first affected " + result[0] + " row");
} catch (SQLException e) {
  conn.rollback();
  throw e;
} finally {
  conn.setAutoCommit(true);
}
```

```
1000 statements, the first affected 1 row
1000 rows, total 500500
```

In each round of the loop the placeholder is filled and `addBatch()` adds the filled statement to the queue. When the loop ends, the queue holds 1000 statements. `executeBatch()` sends them all and returns an array of 1000 elements; each element is 1, because each insert affected one row. The second line of the output was added as a check. The table has 1000 rows; the sum of the values is 1 + 2 + … + 1000 = 1000 · 1001 / 2 = 500,500.

The speed of this code comes from two sources: a single transaction and batched sending. With auto-commit on, every `INSERT` is a separate transaction. At the end of each transaction, SQLite has the data written to disk to guarantee durability and waits for the write to finish. Three measurements were made with a `.db` file on disk (Table 11.24).

Method	Run 1	Run 2	Run 3
One at a time, auto-commit on	1841	2047	1864
`addBatch` + one transaction	7	8	9
One `executeUpdate` at a time + one transaction	2	2	2

Table 11.24: Times to insert 1000 rows into a SQLite file on disk (three runs, ms). The times vary with the machine and the run.

With auto-commit on, 1000 rows took about 2 seconds; with a single transaction, the same job finished in a few milliseconds. The difference is hundreds of times. The last row of the table carries an instructive detail: in SQLite, the main gain comes from the single transaction. Even if the statements are sent one by one, they are as fast as batched sending when they are inside a single transaction. The

reason is that SQLite runs inside the program; sending a statement is just a method call. On a server such as MySQL, each statement needs a network round trip, and `addBatch` also reduces the number of round trips.

Because the whole queue is a single transaction, if one statement fails, `rollback()` undoes the effect of the whole queue. To see this, the constraint `CHECK (value <> 500)` was added to the table and the same code was run:

```
SQLiteException: [SQLITE_CONSTRAINT_CHECK] A CHECK constraint failed (CHECK constraint
failed: value <> 500)
0 rows in the table
```

The 500th statement was rejected, the `catch` block called `rollback()`, and the table stayed empty: all or nothing. After the rollback, the exception is rethrown with `throw e;`; the error is not silently swallowed, and the caller learns that the operation failed. The `finally` block turns auto-commit back on, on success and on failure alike. The standard specifies that a failure in a batch is reported with `BatchUpdateException`, a subclass of `SQLException`. The SQLite driver threw its own `SQLiteException` here. Because both are `SQLException`s, `catch (SQLException e)` catches both cases.

For very large loads, the queue is sent in pieces. For example, `executeBatch` is called after every 500 `addBatch` calls. This way a million statements do not pile up in memory, and memory does not balloon; the transaction can still end with a single `commit`.

11.10 Worked examples and exercises

Example 11.2

On an empty database the following statements run in order. Which rows does the last `SELECT` return?

```
CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT);
INSERT INTO users(name) VALUES ('Ali');
INSERT INTO users(name) VALUES ('Can');
DELETE FROM users WHERE id = 2;
INSERT INTO users(name) VALUES ('Ece');
UPDATE users SET name = 'Efe' WHERE id = 1;
SELECT * FROM users;
```

Ali gets number 1 and Can gets 2. When Can is deleted, the largest `id` in the table becomes 1. Ece gets the largest `id` + 1 = 2: the deleted number has been reused. The `UPDATE` changes only the row with number 1; Ali's name becomes Efe. The result is **(1, Efe)** and **(2, Ece)**. The

statements were verified by running them through JDBC. If the table had been defined with `AUTOINCREMENT`, Ece would have got 3.

Example 11.3

The people program is run a second time with `people.db`. What are the first block of the output (up to the first `---`) and the last block?

The file is persistent. Ayşe (`id` 2) was left from the first run, and `CREATE TABLE IF NOT EXISTS` does not drop the table. The new records get numbers 3 (Ahmet) and 4 (Ayşe) by the largest-`id`-plus-1 rule. `update` makes Ahmet Akdeniz 35. First block: 2 Ayşe Demir 40, 3 Ahmet Akdeniz 35, 4 Ayşe Demir 40. `delete` deletes Ahmet. Last block: 2 Ayşe Demir 40, 4 Ayşe Demir 40. The same person is now recorded twice. The second run of the program really gave this output:

```
2 Ayşe Demir 40
3 Ahmet Akdeniz 35
4 Ayşe Demir 40
---
2 Ayşe Demir 40
4 Ayşe Demir 40
---
```

This result shows a design gap. If the same person should not be added twice, add a constraint such as `UNIQUE(name, surname)` to the table, or have the program check whether the person exists before inserting.

Example 11.4

A login screen is checked with the following code. The table has three people. If the user types `ahmet` as the name and `' OR '1'='1` as the password, what SQL is produced, how many rows come back, and does the login succeed?

```
String sql = "SELECT * FROM people WHERE name = '" + name
           + "' AND password = '" + password + "'";
ResultSet rs = stmt.executeQuery(sql);
boolean loggedIn = rs.next();
```

The concatenation produces this text: `SELECT * FROM people WHERE name = 'ahmet' AND password = '' OR '1'='1'`. In SQL, `AND` is applied before `OR`. The condition reads `(name = 'ahmet' AND password = '') OR '1'='1'`. The right side is true in every row; **3 rows** come

back. `rs.next()` returns `true` and the login **succeeds without the password**. The fix is to use a `PreparedStatement` with `WHERE name = ? AND password = ?`; with the same input, 0 rows come back. All three were verified by running them:

```
SELECT * FROM people WHERE name = 'ahmet' AND password = '' OR '1'='1'
login: true, rows: 3
PreparedStatement login: false
correct password: true
SELECT * FROM people WHERE name = 'ahmet' --' AND password = 'wrong'
login with comment: true
```

The last two lines of the output show another form of attack. If `ahmet' --` is typed in the name field, `--` starts an SQL comment and the password condition is never read; the login succeeds even with a wrong password. In real systems, passwords are not stored as plain text anyway but as hashes.

Example 11.5

If the lines `setAutoCommit(false)`, `commit()`, and `rollback()` were removed from the `transfer` method, what would Ali's and Ece's balances be at the end of the program?

The first transfer succeeds in the same way: Ali 700, Ece 800. In the second transfer, the deposit becomes permanent immediately and Ece rises to 5800. The withdrawal fails because of the `CHECK`, and there is no transaction to roll back. The result: **Ali 700, Ece 5800**. 5000 TL has appeared out of nothing, and the program still prints "Rolled back". The program was run in this form to verify this.

The solutions to the four exercises in the slides were also tried with JDK 21 and the SQLite driver. **The first exercise** fixes the delete program so that it prints a message according to the return value of `executeUpdate`:

```
int n = stmt.executeUpdate(sql);
if (n > 0) System.out.println("Data deleted!");
else System.out.println("No record found.");
```

When the program tried to delete the same row twice, it printed "Data deleted!" first and then "No record found." **In the second exercise**, Ada (60, 70), Bora (30, 50), and Cem (80, 40) are inserted into the `students(id, name, midterm, final)` table in a loop with a `PreparedStatement`. Those whose average (midterm 40 percent, final 60 percent) is 50 or above are listed in descending order:

```
SELECT name, midterm * 0.4 + final * 0.6 AS average FROM students
WHERE midterm * 0.4 + final * 0.6 >= 50 ORDER BY average DESC
```

Ada's average is 60 · 0.4 + 70 · 0.6 = 24 + 42 = 66. Bora's is 12 + 30 = 42, and Cem's is 32 + 24 = 56. Bora is dropped; the output is `Ada 66.0` and `Cem 56.0`. The value is read with `getDouble("average")`. The expression is repeated in `WHERE` because in the logical order `WHERE` comes before `SELECT`, and the alias `average` is not yet defined. `ORDER BY`, however, can use the alias. Because the coefficients are written as decimals such as `0.4`, the result is `REAL`. An integer expression such as `(midterm * 4 + final * 6) / 10`, on the other hand, does integer division in SQLite and drops the fractional part.

The third exercise is a menu-driven program: 1-Add, 2-Update, 3-Delete, 4-List. The menu is built with `Scanner` and `switch`. Because `nextInt()` leaves the end of the line in the buffer, a `nextLine()` is called after it; otherwise the next name read returns an empty string. Update is done with `UPDATE … WHERE id = ?` and delete with `DELETE … WHERE id = ?`. If the number returned by `executeUpdate` is 0, there is no user with that `id`, and "User not found." is printed. **The fourth exercise** saves game scores in a `scores(player, score)` table with `INSERT INTO scores(player, score) VALUES(?, ?)`. The top 10 scores are listed with `SELECT player, score FROM scores ORDER BY score DESC LIMIT ?` and `ps.setInt(1, 10)`. The value of `LIMIT` can also be given with a placeholder; on a table with 12 scores, the query returned 10 rows in descending order.

11.11 Common mistakes

The mistakes mentioned throughout the chapter are collected here with their real messages. Table 11.25 lists JDBC and basic SQL mistakes, and Table 11.26 lists mistakes with relationships, grouping, and transactions. The messages are taken from the `getMessage()` output of SQLite JDBC 3.40.0.0; the driver prefixes each message with an error code such as `[SQLITE_ERROR] SQL error or missing database`, and the table gives only the actual message in parentheses.

Mistake	Example and message
Driver not on the classpath	Jar not added: `No suitable driver found for jdbc:sqlite:sample.db`
Not handling `SQLException`	No `try` or `throws` (compile error): `unreported exception SQLException; must be caught or declared to be thrown`
Table missing or misnamed	Query before the table is created: `no such table: users`
Writing text without quotes	`VALUES(Ahmet, …)` : `no such column: Ahmet`
Concatenating text that contains an apostrophe	`VALUES('Ali's')` : `near "s": syntax error`
The wrong `execute` method	`INSERT` with `executeQuery` : `query does not return ResultSet` (the row is inserted anyway)
Starting column numbers at 0	`rs.getString(0)` : `column 0 out of bounds [1,3]`

Mistake	Example and message
`UPDATE` / `DELETE` without `WHERE`	No error; it changes or deletes all rows (returns 3 on a table of three rows)

Table 11.25: Common mistakes in JDBC and basic SQL.

Mistake	Example and result
Comparing with `= NULL`	`WHERE dept_id = NULL` raises no error and returns 0 rows. The right form is `IS NULL`.
A join without `ON`	`FROM students, depts`: every row matches every row, 4 × 3 = 12 rows.
Ambiguous column name	Both tables have `name` and no table name is given: `ambiguous column name: name`
An aggregate function in `WHERE`	A group condition goes in `HAVING`: `misuse of aggregate: SUM()`
Not enabling foreign key checking	In SQLite, without `PRAGMA foreign_keys = ON`, `dept_id = 9` is silently accepted.
Calling `wasNull()` too late	It looks at the last column read; call it right after the relevant `getInt`.
Selecting an ungrouped column	`SELECT customer, amount … GROUP BY customer`: SQLite returns the amount of one of the rows in the group; in many databases this is an error.
Thousands of `INSERT`s with auto-commit on	Each row is a separate transaction: very slow. Use `setAutoCommit(false)`, `addBatch`, `commit`.
Forgetting `rollback()` on the error path	The `setAutoCommit(true)` in `finally` commits the half-done transaction.

Table 11.26: Common mistakes with relationships, grouping, and transactions.

The run-time lines of the first table were reproduced in a single program:

```
no table -> [SQLITE_ERROR] SQL error or missing database (no such table: users)
unquoted -> [SQLITE_ERROR] SQL error or missing database (no such column: Ahmet)
apostrophe -> [SQLITE_ERROR] SQL error or missing database (near "s": syntax error)
wrong method -> query does not return ResultSet
index 0 -> column 0 out of bounds [1,3]
index 4 -> column 4 out of bounds [1,3]
UPDATE without WHERE -> 3
DELETE without WHERE -> 3
remaining 0
table still exists -> 1
class: org.sqlite.SQLiteException, code: 1, SQLState: null
```

The last line examines the caught exception itself. Its class is `org.sqlite.SQLiteException`, a subclass of `SQLException`. `getErrorCode()` gives SQLite's error code (1, a general SQL error). With most of these mistakes the program compiles and runs; the problem shows up only at run time, and often silently. That is why, after writing JDBC code, you should always check the real contents of the table with DB Browser for SQLite or a `SELECT`.

11.12 Summary

- A database is a persistent, organized, queryable collection of data; a DBMS manages it. Compared with a file, it offers fast search, safe concurrent access, consistency through constraints, column-level permissions (in server DBMSs, for example `GRANT` in PostgreSQL or MySQL; in SQLite, access depends on file permissions), and recovery after a failure.
- The relational model keeps data in tables: a column is an attribute and a row is a record; a primary key identifies each row uniquely. A table resembles a class, a column a field, and a row an object.
- SQLite is an embedded, serverless, single-file relational database. To use it from Java, the driver jar must be on the classpath; the URL has the form `jdbc:sqlite:sample.db`.
- The basic SQL statements cover the CRUD operations: `INSERT`, `SELECT`, `UPDATE`, `DELETE`. `WHERE` filters rows, `ORDER BY` sorts, and `LIMIT` limits. `UPDATE` and `DELETE` without `WHERE` affect all rows.
- If no value is given for `INTEGER PRIMARY KEY`, SQLite assigns the largest `id` + 1; `AUTOINCREMENT` never hands out a number that has been used.
- Normalization keeps each fact in one place and prevents update, insert, and delete anomalies. Tables are linked with foreign keys; in SQLite, checking is turned on with `PRAGMA foreign_keys = ON`.
- `JOIN … ON` returns only matching rows, and `LEFT JOIN` returns all rows of the left table. If `ON` is forgotten, a Cartesian product results.
- `GROUP BY` groups rows; `WHERE` filters rows before grouping, and `HAVING` filters groups after grouping. Logical order: `FROM` → `WHERE` → `GROUP BY` → `HAVING` → `SELECT` → `ORDER BY` → `LIMIT`.
- `NULL` is an unknown value; it is checked with `IS NULL`. `getInt` returns 0 for `NULL`; to tell the difference, call `wasNull()` right afterward.
- An index turns a scan into a B-tree search; the cost is that the index must be updated on every write as well.
- The JDBC chain: `DriverManager` → `Connection` → `Statement` / `PreparedStatement` → `ResultSet`. These are interfaces; the driver implements them.
- `execute` returns a `boolean` and is used for table operations, `executeUpdate` returns the number of rows affected, and `executeQuery` returns a `ResultSet`. The result is read with `while (rs.next())`; column numbers start at 1.
- User input is never concatenated into the SQL text; it is always passed with `?` and `setXxx`. Otherwise SQL injection results.
- A DAO collects a table's SQL in a single class and returns rows as objects.
- Transaction: `setAutoCommit(false)`, `commit()`, and on error `rollback()`; all or nothing (ACID). Bulk inserts are done in a single transaction.

- Resources are opened with try-with-resources and closed in the order `rs`, `stmt`, `conn`. `SQLException` is a checked exception.

11.13 Key terms

database relational model primary key foreign key query normalization join aggregate function database index result set prepared statement SQL injection data access object (DAO) transaction batch update checked exception try-with-resources

---

11.14 Test Yourself

The questions use the chapter's three tables: `students` (Ada 1, Bora 2, Cem 1, Deniz `NULL`), `depts` (CS, EE, ME), and `orders` (Ali 120, Ece 80, Ali 50, Can 200, Ali 30, Ece 40). The answers were verified by running them with JDK 21 and SQLite JDBC.

1. What does `SELECT COUNT(*) FROM students WHERE dept_id <> 2` return?

   Answer

   2. Ada and Cem (1) satisfy the condition; Bora (2) does not. Deniz's `dept_id` is `NULL`, and `NULL <> 2` is not true but unknown; Deniz is not counted.

2. Three rows (1, 2, 3) are inserted into a table with the column `id INTEGER PRIMARY KEY`, the row with number 2 is deleted, and a new row is inserted. What are the numbers in the table?

   Answer

   1, 3, 4. The new row gets the largest `id` + 1 = 4. The gap in the middle is not filled; only if the largest number had been deleted would that number have been reused.

3. What does `SELECT customer, SUM(amount) FROM orders WHERE amount < 100 GROUP BY customer HAVING COUNT(*) >= 2` return?

   Answer

   `WHERE` first drops the orders of 120 and 200. Ali's 50 and 30 and Ece's 80 and 40 remain; Can has no orders left. Both groups have a count of 2. The result: **Ali 80** and **Ece 120**.

4. What does `SELECT COUNT(*) FROM depts d LEFT JOIN students s ON s.dept_id = d.id`

return? What if `LEFT` is removed?

> **Answer**
>
> CS matches two students and EE one; ME matches none, but `LEFT JOIN` keeps it with one row: **4**. With a plain `JOIN`, ME drops out: **3**. Deniz is in neither query, because the preserved side is the departments.

5. A name search is done with `"SELECT name FROM people WHERE name = '" + input + "'"`. If the user types `' OR 1=1 --`, how many rows come back on a table of three people?

> **Answer**
>
> The SQL text becomes `SELECT name FROM people WHERE name = '' OR 1=1 --'`. `--` turns the final quotation mark into a comment; the condition is true in every row. **3** rows come back. With a `PreparedStatement`, 0 rows would come back.

6. The supplies table has stapler 40, scissors 10, and tape 25. The statements `INSERT … ('pen', 5)`, `UPDATE supplies SET stock = stock + 10 WHERE stock < 20`, and `DELETE FROM supplies WHERE stock > 100` are added to a `Statement` in that order with `addBatch`, and then `executeBatch()` is called. What is the returned array?

> **Answer**
>
> **[1, 2, 0]**. The insert affects one row. In the update, the rows with stock below 20 are scissors (10) and the newly inserted pen (5); they become 20 and 15. There is no row to delete. The `addBatch(String)` form of `Statement` can also queue different SQL texts.

7. The `students` table is read in `id` order; for each row, `total += rs.getInt("dept_id")` is executed, and `none++` if `rs.wasNull()` is true. At the end, what are `total` and `none`?

> **Answer**
>
> `getInt` gives 0 for Deniz: 1 + 2 + 1 + 0 = **4**. `wasNull()` is true only for Deniz: `none` = **1**.

8. The table `members(id INTEGER PRIMARY KEY, email TEXT UNIQUE)` is empty. `setAutoCommit(false)` is called, `a@x` and `b@x` are inserted, then `rollback()`, then `c@x` is inserted, then `commit()`. Which rows are in the table?

> **Answer**
>
> A single row: **(1, c@x)**. `rollback()` undoes the first two inserts; undone inserts do not use up numbers either. `c@x` is inserted into the empty table and gets 1.

11.15 Further reading

- Oracle, *The Java Tutorials*, "JDBC Database Access": the official, free source that explains connecting, running queries, `PreparedStatement`, and transactions with short examples.
- SQLite documentation, "Datatypes In SQLite" and "Appropriate Uses For SQLite": explains SQLite's flexible type system and when a server-based database is more suitable.
- OWASP, *SQL Injection Prevention Cheat Sheet*: describes the kinds of injection and why prepared statements are the primary defense, with examples.
- C. S. Horstmann, *Core Java, Volume II: Advanced Features* (12th ed.), chapter "Database Programming": a detailed treatment of JDBC, transactions, and advanced features of result sets.

11.16 References

1. Oracle, *The Java Tutorials*: JDBC Database Access (Establishing a Connection; Processing SQL Statements with JDBC; Using Prepared Statements; Using Transactions).
2. Oracle, *Java SE 21 API Documentation*: the `java.sql` package (`DriverManager`, `Connection`, `PreparedStatement`, `ResultSet`).
3. SQLite documentation, sqlite.org: *SQL As Understood By SQLite*; *Datatypes In SQLite*; *Appropriate Uses For SQLite*.
4. Xerial, *SQLite JDBC Driver*, github.com/xerial/sqlite-jdbc.
5. P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson (Accessing Databases with JDBC).
6. C. S. Horstmann, *Core Java, Volume II: Advanced Features*, 12th ed., Pearson (Database Programming).
7. OWASP, *SQL Injection Prevention Cheat Sheet*.

APPENDIX A

Glossary of Terms

The terms used in the book, each with a short definition. The number next to a term is the chapter where it is explained in detail. A searchable version of the glossary is on the course page.

A

acceptance testing 7
Testing that runs the system with users on real data to decide whether the customer will accept it.

ACID 11
The four properties of a transaction: atomicity, consistency, isolation, and durability.

adapter class 9
A ready-made class that implements every method of a listener interface with an empty body; a subclass overrides only the method it cares about.

aggregate function 11
A function such as COUNT, SUM, AVG, MIN, or MAX that produces a single value from a set of rows.

amortized cost 5
The average cost of occasionally expensive operations over a long sequence of calls; appending to an ArrayList takes constant time on average.

assertion 7
A statement that compares an expected result with the actual one or checks an assumption.

atomic operation 10
An operation that completes in a single indivisible step (AtomicInteger).

auto-commit 11
The default mode of a JDBC connection: each statement is its own transaction and becomes permanent as soon as it runs.

AWT (Abstract Window Toolkit) 9
Java's first graphics library (java.awt); today used mostly for Graphics, Color, Font, layout managers, and event classes.

B

batch update 11
Collecting many statements with addBatch and sending them together with executeBatch.

binary heap 5
A complete binary tree stored in an array in which every parent is less than or equal to its children; the internal structure of PriorityQueue.

black-box / white-box testing 7
Testing done only with inputs and expected outputs without looking inside the code, versus testing that uses knowledge of the code's structure to run every branch and loop at least once.

boundary value analysis 7
Choosing the edge values of each equivalence class and the first values of the invalid region as test inputs.

breakpoint 7
A line at which the debugger stops the program.

buffer / buffering 8
Accumulating data in memory and reading or writing it in bulk to reduce disk accesses.

busy waiting 10
Waiting for a condition by testing it continuously in a loop; it keeps a core busy for nothing.

byte stream / character stream 8
The InputStream/OutputStream and Reader/Writer class families.

C

capacity 5
The total number of cells in the internal array of a dynamic array; not the same as size(), which counts the filled cells.

channel (NIO) 8
In NIO, a connection that reads and writes through a buffer in both directions.

character encoding 8
The rule that determines which byte sequences are used to store characters.

checked exception 6
An exception that the compiler requires you to catch or declare.

code coverage 7
The proportion of code lines that the tests execute.

collection 5
A data structure that holds multiple objects together.

Collections Framework 5
Java's collection library of interfaces, classes, and algorithms.

collision 5
Two different elements falling into the same bucket of a hash table; colliding elements form a chain in the bucket.

comparator 5
An object that compares two objects from the outside and defines an alternative ordering.

compare-and-set (CAS) 10
A single atomic operation that writes a new value and returns true if the value is still the expected one, and otherwise changes nothing and returns false.

component 9
A visual element placed in a window (JButton, JLabel, and so on).

concurrency / parallelism 10
Tasks making progress in an interleaved way over time versus actually running at the same instant.

container 9
A component that contains other components (JFrame, JPanel).

content pane 9
The inner area of a window below the title bar; components added with add are placed here.

context switching 10
The processor switching from one thread to another; the state of the running thread (program counter, registers, stack pointer) is saved and that of the next one is loaded.

continuous integration (CI) 7
Systems (GitHub Actions, Jenkins) that run the tests automatically without the programmer having to ask.

critical section 10
A piece of code that accesses shared data and that only one thread may execute at a time.

custom exception 6
An application-specific exception derived from Exception or RuntimeException.

D

daemon thread 10
A helper thread that does not prevent the JVM from shutting down.

data access object (DAO) 11
A design that gathers the SQL operations of a table in one class and returns objects.

database 11
A system that stores data in an organized, persistent, and queryable way.

database index 11
An extra structure that keeps the values of a column in order to speed up searching.

deadlock 10
Threads waiting forever for locks held by one another.

debugger 7
A tool that runs a program step by step and lets you watch its variables.

debugging 7
The process of finding the cause of a bug and fixing it.

decorator (wrapper stream) 8
A stream that wraps another stream and adds a capability to it, such as buffering.

defect / failure 7
The wrong line of code that reflects the programmer's mistake versus the wrong behavior that appears when that code runs (wrong output, a crash, a hang).

deque (double-ended queue) 5
A queue that allows insertion and removal at both ends.

doubly linked list 5
A list in which each node holds a value and references to the previous and next nodes (LinkedList).

dynamic array 5
An array that grows by copying into a larger array when it becomes full (ArrayList).

E

equivalence class 7
A set of inputs that cause the same behavior in the program; choosing one representative from each class is enough.

error (Error class) 6
Serious problems that a program usually cannot recover from (OutOfMemoryError).

event 9
A user action such as a click or key press, or a notification from the system.

Event Dispatch Thread (EDT) 9
The single thread on which Swing components are built and events are handled.

event listener 9
An interface or object that defines the methods to be called when an event occurs.

event-driven programming 9
A program whose flow is determined by the order in which events arrive rather than by the order of lines in the code.

exception 6
An event that disrupts the normal flow while the program runs and is thrown as an object.

exception propagation 6
The progress of an uncaught exception up the call stack toward the caller.

F

file pointer 8
A pointer in a RandomAccessFile that marks where reading and writing happen and advances by the number of bytes processed at each operation.

finally block 6
A block that runs whether or not an exception occurs.

foreign key 11
A column that refers to the primary key of another table.

G

garbage collector 7
The JVM component that reclaims the memory of objects that no reference points to.

generics 5
Writing classes and methods with a type parameter (List<String>).

graphical user interface (GUI) 9
An interface that provides visual interaction through elements such as windows, buttons, and menus.

H

happens-before 10
A transitive relation guaranteeing that if operation A happens-before B, everything A wrote is visible in B.

hashing / hash table 5
Placing an element in the bucket computed from its hash code to get O(1) access on average.

I

integration test 7
A test that exercises components working together with their real dependencies.

iterator 5
An object that gives the elements of a collection in turn and allows removal while traversing.

J

JDBC (Java Database Connectivity) 11
Java's standard API for connecting to databases (java.sql); the code that talks to a specific database is supplied by that database's JDBC driver.

JDBC driver 11
The library that lets Java talk to a specific database (the sqlite-jdbc jar for SQLite).

join 11
The SQL operation that matches the rows of two tables according to an ON condition.

L

layout manager 9
An object that determines how components are arranged in a container.

lazy evaluation 5
Stream intermediate operations processing no elements and only recording a recipe; elements are processed when a terminal operation is called.

linked list 5
A list in which each element holds a link to the next one (LinkedList).

list 5
An ordered, indexed collection that allows duplicates.

lock 10
An object that gives a single thread access to a resource (ReentrantLock).

logic error 6
An error in which the program runs but produces a wrong result.

lost update 10
The loss of one of two increments when two threads read the same old value and write over it.

M

magic number 8
The leading bytes that identify the type of a file; every .class file starts with CA FE BA BE.

main thread 10
The thread the JVM starts to run the main() method.

map 5
A structure that holds key-value pairs in which each key is unique.

marker interface 8
An interface with no methods whose only job is to mark a class (Serializable).

memory leak 7
Memory that is not freed because references to objects that are no longer needed remain.

monitor 10
The lock and waiting mechanism attached to every object, acquired with synchronized.

multithreading 10
A program running with several threads.

mutual exclusion 10
Only one thread being in the critical section at a time.

N

natural ordering (Comparable) 5
The default ordering that a class defines with compareTo.

normalization 11
Splitting data into tables so that each fact is stored in only one place, to avoid repetition.

O

one-to-many / many-to-many relationship 11
A relationship in which one record matches several records in the other table (one department, many students) versus one in which many records match in both directions and which needs a junction table.

P

prepared statement 11
A SQL statement whose parameters are supplied separately with ?, which prevents injection.

primary key 11
A column that uniquely identifies each row of a table.

priority queue 5
A queue that gives the element with the highest priority first.

process 10
A running program that has its own memory space.

producer–consumer problem 10
The coordination of threads that produce into and consume from a shared buffer.

profiler 7
A tool that measures where a program spends its time and memory.

property-based testing 7
Testing that checks with random inputs that a property always holds.

Q

query 11
A SQL command that requests data from the database (SELECT).

queue 5
A structure that works on the first-in, first-out (FIFO) principle.

R

race condition 10
An outcome that depends on the order in which threads run.

random access file 8
A file that can be read and written by moving the pointer to the desired position.

red-black tree 5
The tree structure TreeSet uses internally to keep its elements sorted.

reentrant lock 10
A lock that lets the thread holding it acquire the same lock again without waiting; it keeps a hold count.

refactoring 7
Cleaning up code (removing duplication, fixing names) while the tests stay green; the last step of the TDD cycle.

regression test 7
Rerunning the old tests after a change.

relational model 11
A model that stores data in tables made of rows and columns.

result set 11
An object that lets you read the rows returned by a query one by one.

runtime error 6
An error that appears while the program runs and leads to an exception.

S

scheduler 10
The JVM and operating-system component that decides which thread runs when and on which core; the programmer cannot control this order.

selector (NIO) 8
An NIO object that monitors several channels with a single thread.

semaphore 10
A counter that lets at most n threads access a resource at the same time.

serialization 8
Converting an object into a byte sequence to store it and reading it back.

set 5
A collection that does not allow duplicate elements.

shuffle 5
Randomly changing the order of elements (Fisher-Yates).

spurious wakeup 10
A thread in wait waking up without any notify; for this reason the condition is checked with while, not if.

SQL injection 11
An attack in which user input gets mixed into a SQL command and changes it.

stable sort 5
A sort that preserves the relative order of elements considered equal (TimSort, used by Collections.sort).

stack 5
A structure that works on the last-in, first-out (LIFO) principle.

stack trace 6
The output that shows the type and message of an exception and the chain of calls.

stack unwinding 6
Moving up the call stack while looking for a suitable catch and discarding the frames along the way.

standard input / output 8
The program's keyboard (System.in) and screen (System.out) streams.

static analysis 7
An examination that searches for bug patterns without running the code (SpotBugs).

stream (I/O) 8
A sequence of bytes or characters flowing in order from a source to a destination.

synchronization 10
Coordinating the access of threads to shared data.

T

test double 7
A simple object that stands in for an external dependency in a unit test by imitating it.

test-driven development (TDD) 7
A method of writing the test first and then the code that makes it pass.

text file / binary file 8
A file whose bytes are read as characters versus a file that holds raw data.

thread 10
An independently progressing flow of execution inside a program.

thread pool 10
A structure that distributes tasks to threads that are waiting ready (ExecutorService).

thread safety 10
The ability of a class to be used correctly from several threads at the same time.

thread states 10
NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED.

throw / catch 6
Creating and sending an exception, and handling it in a catch block.

transaction 11
A group of SQL statements executed on an all-or-nothing basis.

try-catch block 6
A structure that puts code that may raise an exception in try and the code that handles it in catch.

try-with-resources 6
A form of try that automatically closes opened resources when the block ends.

type safety 5
Catching values of the wrong type at compile time.

U

unchecked exception 6
An exception derived from RuntimeException that the compiler does not check.

unit test 7
An automated test that exercises a single method or class on its own.

V

virtual thread 10
A thread managed by the JVM rather than the operating system, of which millions can be created (Java 21); suited to many short tasks that mostly wait.

APPENDIX B

Prerequisites for Object-Oriented Programming with Java

JAVA · PREREQUISITES

The course starts from how a computer works and soon moves on to writing, compiling, and running Java programs. The 28 topics on this page gather what the course builds on in six groups: computer fundamentals, number systems, algorithms and flowcharts, basic programming concepts, the tools needed to write Java, and two habits of thought behind objects.

Each card has a short explanation, a small example, links to the slides where the topic is used, and a check question; click the question to open its answer.

Computer fundamentals (3) **Number systems and data representation (4)**

Algorithms and flowcharts (6) **Programming concepts (7)**

Tools and setup (6) **Getting ready for objects (2)**

Computer fundamentals

What happens inside the machine while a program is stored, loaded into memory, and run.

CPU

Hardware: processor, memory, and storage

CPU · RAM · disk · input/output

A program sits on the disk as a file. To run, it is loaded into main memory (RAM); the processor then executes the program's instructions one by one, reading and writing data in memory. RAM is fast but loses its contents when the power is cut, while the disk keeps storing files.

WHY IT MATTERS

While a Java program runs, its variables, objects, and references live in memory; after the program ends, only what was written to a file remains.

WHERE IT IS USED **Chapter 1.1 · Basic Parts** **Chapter 1.1 · CPU – Memory – Bus**

Chapter 1.1 · Memory Hierarchy

CHECK QUESTION

▶ **A program executes int score = 90; and then the computer is shut down. Is the value 90 still there the next day?**

EXAMPLE (JAVA)

```
Runtime rt = Runtime.getRuntime();
System.out.println(rt.availableProcessors());          // for example 8 cores
System.out.println(rt.maxMemory() / (1024 * 1024) + " MB"); // memory the JVM
may use
```

/

Files, folders, and paths

absolute · relative · extension

A path names a file: an absolute path starts at the root (C:\Users\ann\course\Hello.java), while a relative path starts at the current working directory (course\Hello.java). The extension tells the type of file: .java is source code, .class is compiled bytecode. Windows hides known extensions by default, so a file can end up named Hello.java.txt without anyone noticing.

WHY IT MATTERS

javac and java are run in the folder that holds the source file; file input and output also open files with relative or absolute paths.

WHERE IT IS USED **Chapter 2.2 · .java Files** **Chapter 8 · The java.io.File Class**

Chapter 8 · Files and Directories with File

CHECK QUESTION

▶ **The working directory is C:\course and a program opens a file with new File("grades.txt"). Which file is it?**

EXAMPLE (PATHS)

```
C:\Users\ann\course\Hello.java    absolute path (Windows)
/home/ann/course/Hello.java       absolute path (Linux, macOS)
course\Hello.java                 relative path from the C:\Users\ann
directory
Hello.java                        source code
Hello.class                       bytecode produced by javac
```

01 Programs, compilers, and interpreters

source code · machine code · bytecode

A processor executes only machine code. A compiler translates the whole program in advance; an interpreter reads and executes the program step by step. Java combines the two: javac compiles source code into platform-independent bytecode, and the Java Virtual Machine (JVM) interprets this bytecode and, while the program runs, compiles the frequently used parts into machine code.

WHY IT MATTERS

The difference between a compile-time error and a run-time error, and the roles of the javac and java commands, come from this two-step process.

WHERE IT IS USED **Chapter 2.1 · Compiled and Interpreted Languages**

Chapter 2.2 · Compilation **Chapter 2.2 · Abstraction Levels**

CHECK QUESTION

▶ **A Hello.class file was produced on Windows. Can this file be run on Linux?**

EXAMPLE (TWO STEPS)

```
Hello.java  --javac-->  Hello.class  --java (JVM)-->  program output
(source code)             (bytecode)
```

Number systems and data representation

How numbers and characters are stored as bits, and the surprising results this has.

2 Binary, decimal, and hexadecimal systems

base 2 · base 10 · base 16

In base 2 each digit is worth a power of two: $1101_2 = 8 + 4 + 0 + 1 = 13$. Hexadecimal packs four bits into a single digit from 0–9 and A–F, so $1111\ 1111_2 = FF_{16} = 255$. You should be able to convert a number to binary by repeatedly dividing by 2 and convert back by adding up place values.

WHY IT MATTERS

Bit patterns, two's complement representation, color values, and character codes are written in these bases; Java also accepts binary and hexadecimal literals.

WHERE IT IS USED **Chapter 1.1 · Place Value** **Chapter 1.1 · To Binary: Trace**

Chapter 1.1 · Bases in Java

CHECK QUESTION

▶ **What are 1011_2 and $2A_{16}$ in decimal?**

EXAMPLE (JAVA)

```
System.out.println(Integer.toBinaryString(13));   // 1101
System.out.println(Integer.toHexString(255));      // ff
System.out.println(0b1101 + " " + 0x1F);           // 13 31
System.out.println(Integer.parseInt("101", 2));    // 5
```

± Bits, bytes, and integer ranges

byte · int · long · two's complement · overflow

n bits can hold 2^n different values. Java stores signed integers in two's complement form, so a byte (8 bits) takes values from −128 to 127, and an int (32 bits) takes values from about −2.1 billion to 2.1 billion. When a result does not fit the type, no error occurs; the value silently wraps around.

WHY IT MATTERS

Choosing between int and long, and noticing overflow in factorials, sums, and products, depend on knowing these limits.

WHERE IT IS USED **Chapter 1.1 · Two's Complement**

Chapter 1.1 · Two's Complement and Overflow **Chapter 2.2 · Integer Types**

Chapter 2.2 · Overflow

CHECK QUESTION

▶ **How many different values can be stored in 8 bits, and what is the range of Java's byte type?**

EXAMPLE (JAVA)

```
byte b = (byte) 130;
System.out.println(b);                 // -126
int max = Integer.MAX_VALUE;           // 2147483647
System.out.println(max + 1);           // -2147483648 (overflow)
System.out.println((long) max + 1);    // 2147483648
```

.1 Fractional numbers are approximate

double · rounding error · integer division

A double stores a binary fraction, so most decimal fractions such as 0.1 can be held only approximately, and small errors appear in sums. For this reason two double values are compared not with == but with a tolerance. The quotient of two integers is also an integer: the result of 7 / 2 is 3, while the result of 7 / 2.0 is 3.5.

WHY IT MATTERS

If these two effects are not taken into account, averages, unit conversions, and money calculations silently give wrong results.

WHERE IT IS USED **Chapter 1.1 · Floating Point** **Chapter 2.2 · Floating-Point Types**

Chapter 2.2 · Floating-Point Precision **Chapter 2.2 · Integer Division and Conversion**

CHECK QUESTION

▶ **Why are money amounts often stored as whole numbers of cents?**

EXAMPLE (JAVA)

```
System.out.println(0.1 + 0.2);                          // 0.30000000000000004
System.out.println(0.1 + 0.2 == 0.3);                   // false
System.out.println(Math.abs(0.1 + 0.2 - 0.3) < 1e-9);   // true
System.out.println(7 / 2 + " " + 7 / 2.0);              // 3 3.5
```

A Characters: ASCII and Unicode

char · character code · UTF-8

Every character is stored as a number: in ASCII, 'A' is 65, 'a' is 97, and '0' is 48. Unicode extends this numbering to all writing systems; letters of other alphabets have codes too (for example, é is 233). Java's char holds a 16-bit Unicode unit and can be used in arithmetic. Files usually store text in the UTF-8 encoding.

WHY IT MATTERS

Character arithmetic such as converting the character '7' to the number 7, comparing strings, and reading non-English text from a file correctly all rest on character codes.

WHERE IT IS USED **Chapter 1.1 · ASCII Table** **Chapter 1.1 · Unicode**

Chapter 2.2 · char and int **Chapter 1.1 · UTF-8**

CHECK QUESTION

▶ **The code of the character 'a' is 97. What does System.out.println('a' + 1); print?**

EXAMPLE (JAVA)

```
char c = 'A';
System.out.println((int) c);          // 65
```

```
System.out.println((char) (c + 2));   // C
System.out.println('7' - '0');        // 7
System.out.println((int) 'é');        // 233
```

Algorithms and flowcharts

Planning the solution before writing code, and tracing code by hand.

→ Algorithm: input, processing, output

finite · precise · terminating

An algorithm is a sequence of a finite number of precise steps that transforms the given input into the desired output and always halts. Before coding, the problem is defined clearly: what is given, what must be produced, and which special cases exist (zero, negative numbers, an empty list).

WHY IT MATTERS

Every exercise and example program in the course starts from an algorithm; most mistakes come not from Java syntax but from a vague problem definition.

WHERE IT IS USED **Chapter 1.3 · What Is an Algorithm?**

Chapter 1.2 · Input, Process, Output **Chapter 1.3 · Defining the Problem**

CHECK QUESTION

▶ **“Keep adding 1 to x until x is negative.” If x starts at 5, is this an algorithm?**

EXAMPLE (PSEUDOCODE)

```
Read a, b
If a > b then
    max ← a
Otherwise
    max ← b
Write max
```

≡ From pseudocode to code

pseudocode · step-by-step translation

Pseudocode describes the steps of an algorithm in plain, orderly language without worrying about syntax. Then a Java counterpart is written for each line: Read becomes reading with Scanner, ← becomes an assignment, Loop becomes a loop, and Write becomes a System.out.println call.

WHY IT MATTERS

Writing the steps first separates thinking about the solution from fighting compiler errors.

WHERE IT IS USED **Chapter 1.3 · Pseudocode** **Chapter 1.3 · Implementation**

Chapter 1.3 · Addition Program Trace

CHECK QUESTION

▶ **What is printed in the example for n = 4?**

EXAMPLE (PSEUDOCODE AND JAVA)

```
sum ← 0                      int sum = 0;
Loop i = 1 to n              for (int i = 1; i <= n; i++) {
    sum ← sum + i                sum = sum + i;
                             }
Write sum                    System.out.println(sum);
```

◇ Flowcharts and the three control structures

sequence · selection · repetition

A flowchart draws an algorithm with standard shapes: an oval for start and end, a parallelogram for input and output, a rectangle for computation, and a diamond for a decision. Every algorithm can be built from three structures: sequence, selection (if), and repetition (loop).

WHY IT MATTERS

When the flow of a loop or of nested decisions is drawn, it is easier to see where the condition is tested and when the program leaves the loop.

WHERE IT IS USED **Chapter 1.4 · Basic Symbols** **Chapter 1.4 · Symbols and Java**

Chapter 1.4 · Control Structures

CHECK QUESTION

▶ **In which flowchart shape is a condition such as n > 0 written, and how many exits does that shape have?**

EXAMPLE (SHAPES AND JAVA)

```
( start / end )      oval            program starts / ends
/ input, output /    parallelogram   Scanner, System.out.println
[ computation ]      rectangle       sum = sum + i;
< condition? >       diamond         if (n > 0), while (i <= n)
```

≣

Tracing code by hand

trace table · variable values step by step

Tracing means running a program on paper: a table records the value of every variable after each step. It is the most reliable way to predict the output of a loop, to find an off-by-one error, and to understand someone else's code.

WHY IT MATTERS

Many slides and exam questions ask for the output of a short piece of code; a trace table lets you answer these without a computer.

WHERE IT IS USED **Chapter 1.3 · Addition Program Trace**

Chapter 3.1 · Summing Numbers (for) **Chapter 1.2 · Iteration**

CHECK QUESTION

▶ **int x = 5; while (x > 1) x = x / 2; How many times does the loop body run, and what is x at the end?**

EXAMPLE (TRACE TABLE)

```
int s = 0;
for (int i = 1; i <= 4; i++) s += i;

//  i :  1   2   3   4
//  s :  1   3   6  10     -> the loop ends when i becomes 5
```

log

Logarithms and growth rates

$\log_2 n \cdot n \cdot n \log n \cdot n^2 \cdot 2^n$

$\log_2 n$ tells to which power 2 must be raised to get n: since $2^{10} = 1024$, $\log_2 1024 = 10$. Halving a number again and again brings it down to 1 in about $\log_2 n$ steps; that is why the logarithm grows very slowly: when n grows a thousandfold, $\log_2 n$ grows by only about 10. As the input size n grows, different functions grow at very different rates; from small to large the order is 1, log n, n, n log n, n^2, 2^n. With n^2 growth, doubling n quadruples the work; with 2^n growth, even increasing n by 1 doubles the work.

WHY IT MATTERS

The costs of searching and sorting algorithms and of array and collection operations are compared using notations such as O(1), O(log n), O(n), O(n log n), and $O(n^2)$. The exponentially growing number of calls in plain recursive Fibonacci is explained by the same idea.

WHERE IT IS USED **Chapter 3.3 · Complexity: O(1), O(n), $O(n^2)$**

Chapter 1.3 · Linear vs. Binary **Chapter 5 · Comparing Map Classes**

CHECK QUESTION

▶ **In a sorted array the search space is halved at each step. If the array has 1,000,000 elements, what is the maximum number of steps needed?**

EXAMPLE (JAVA AND TABLE)

```
int n = 1000, steps = 0;
while (n > 1) {
    n = n / 2;                 // halved on every round
    steps++;
}
System.out.println(steps);    // 9  (log₂ 1000 ≈ 9.97)

//  n       log₂ n   n log₂ n    n²
//  10      ≈ 3.3    ≈ 33        100
//  1000    ≈ 10     ≈ 10,000    1,000,000
```

n! Factorial, summation notation, and inductive definitions

n! · Σ · C(n, k) · base value and rule

n! (n factorial) is the product of the numbers from 1 to n, and 0! = 1 by convention: 5! = 120. The symbol Σ writes a sum compactly: the sum of i from 1 to n is n(n + 1)/2. A sequence can also be defined inductively: one or several starting values are given, and each term is found from the earlier ones by a rule. n! = n · (n − 1)! and F(n) = F(n − 1) + F(n − 2) are such definitions. The number of ways to choose k items out of n is the combination C(n, k) = n! / (k! (n − k)!); in Pascal's triangle each number is the sum of the two numbers above it: C(n, k) = C(n − 1, k − 1) + C(n − 1, k).

WHY IT MATTERS

A recursive method turns such a definition directly into code: the starting value becomes the base case and the rule becomes the recursive step. The factorial, Fibonacci, Catalan, and Bell sequences and Pascal's triangle are given with this notation.

WHERE IT IS USED **Chapter 3.5 · Base Case and Recursive Step**

Chapter 3.5 · Catalan Series **Chapter 3.3 · Ragged Array: Code Trace**

CHECK QUESTION

▶ **What is the term a(4) of the sequence defined by a(0) = 3 and the rule a(n) = a(n − 1) + 2?**

EXAMPLE (A PREVIEW OF JAVA)

```
static long factorial(int n) {
    if (n == 0) return 1;              // base: 0! = 1
```

```
    return n * factorial(n - 1);       // rule: n! = n · (n − 1)!
}
static long combination(int n, int k) {
    if (k == 0 || k == n) return 1;    // the edges of the triangle
    return combination(n - 1, k - 1) + combination(n - 1, k);
}
// factorial(5) == 120, combination(5, 2) == 10
```

Programming concepts

The basic building blocks of every program, whatever the language.

x

Variables, types, and assignment

int · double · boolean · char · String

A variable is a named place in memory whose type determines which values it can hold. Java checks types at compile time: an int cannot hold "5", and a variable must be given a value before it is read. The = sign is not mathematical equality but assignment (the value on the right side is copied into the variable).

WHY IT MATTERS

Every program keeps its data in variables; primitive types and reference types later behave differently when they are assigned or passed to methods.

WHERE IT IS USED **Chapter 2.2 · Variables** **Chapter 2.2 · Assignment Statements**

Chapter 2.2 · Primitive Data Types

CHECK QUESTION

▶ **int a = 3; int b = a; a = 10; What is b?**

EXAMPLE (JAVA)

```
int age = 20;
double gpa = 3.25;
boolean passed = gpa >= 2.0;
String name = "Ann";
age = age + 1;
System.out.println(name + " " + age + " " + passed);   // Ann 21 true
```

%

Expressions and operators

/ · % · operator precedence · string concatenation

Arithmetic follows precedence rules: * / % are performed before + −, and operators of equal precedence are applied from left to right. The remainder operator % together with integer division / splits a number into its digits. When one of the operands of + is a String, + does not add but concatenates the texts.

WHY IT MATTERS

Digit sums, odd/even tests, divisibility, and formatted output rest on these operators.

WHERE IT IS USED **Chapter 2.2 · Arithmetic Operators** **Chapter 2.2 · Precedence**

Chapter 3.4 · Concatenation Order: Code Trace

CHECK QUESTION

▶ **For n = 4729, what are n % 10 and n / 10?**

EXAMPLE (JAVA)

```
System.out.println(17 / 5);        // 3
System.out.println(17 % 5);        // 2
System.out.println(2 + 3 * 4);     // 14
System.out.println("1" + 2 + 3);   // 123
System.out.println(1 + 2 + "3");   // 33
```

&& Logical expressions

&& · || · ! · short circuit · De Morgan

Conditions are combined with && (and), || (or), and ! (not). Java evaluates them from left to right and stops as soon as the result is known (short-circuit evaluation); so a later part can rely on an earlier check. De Morgan's laws distribute a negation inward: !(a && b) is the same as !a || !b.

WHY IT MATTERS

Range checks, input validation, and loop conditions are boolean expressions; a misplaced && or || is among the most common logic errors.

WHERE IT IS USED **Chapter 2.2 · Logical Operators** **Chapter 2.2 · Short-Circuit**

Chapter 3.1 · Nested Decisions

CHECK QUESTION

▶ **How is !(x >= 1 && x <= 10) written without using !?**

EXAMPLE (JAVA)

```
int x = 7;
boolean inRange = x >= 1 && x <= 10;      // true
```

```
boolean outside = !(x >= 1 && x <= 10);   // false: same as x < 1 || x > 10
int d = 0;
if (d != 0 && 10 / d > 1)                 // the right side is skipped: no
division by zero
    System.out.println("big");
System.out.println(inRange + " " + outside);   // true false
```

? Decisions: if, else if, switch

branching · nested decisions

if runs a block if its condition is true; else provides the alternative. An else-if chain tests the conditions in order and runs only the first branch whose condition holds. switch picks a case according to the value of a single expression.

WHY IT MATTERS

Grade calculation, menus, input checks, and almost every algorithm in the course make decisions.

WHERE IT IS USED **Chapter 3.1 · The if-else Statement** **Chapter 3.1 · else-if Chains**

Chapter 3.1 · The switch-case Statement

CHECK QUESTION

▶ **int x = 5; if (x > 3) print "A"; else if (x > 1) print "B"; What is printed?**

EXAMPLE (JAVA)

```
int score = 72;
if (score >= 85) System.out.println("A");
else if (score >= 70) System.out.println("B");   // B
else System.out.println("C");
```

↻ Loops: for, while, do-while

counter · condition · nested loops

A loop repeats a block as long as its condition is true. for suits cases where the number of repetitions is known, while suits a condition tested before every round, and do-while suits cases where the body must run at least once. Every loop needs a variable that changes in such a way that the condition eventually becomes false.

WHY IT MATTERS

Sums, searches, patterns, input loops, and array operations are written with loops; infinite loops and loops that run one time too few or too many are frequent mistakes.

WHERE IT IS USED **Chapter 3.1 · The for Loop** **Chapter 3.1 · The while Loop**

Chapter 3.1 · Comparing the Loop Types **Chapter 3.1 · Nested Loops**

CHECK QUESTION

▶ **How many times does the loop for (int i = 10; i > 0; i -= 3) run?**

EXAMPLE (JAVA)

```
for (int i = 0; i < 3; i++) System.out.print(i + " ");   // 0 1 2
System.out.println();
int n = 1234, digits = 0;
while (n > 0) {
    n /= 10;
    digits++;
}
System.out.println(digits);                               // 4
```

f()

Functions: parameters and return values

method · parameter · return · void

A function (a method in Java) gathers a computation under one name: it receives values through parameters and can return a single result with return; a void method returns no value. Java passes arguments by value; so a method cannot change the caller's int variable.

WHY IT MATTERS

Programs are divided into methods from the very beginning; classes, too, are structures that bring methods and data together.

WHERE IT IS USED **Chapter 3.2 · Defining a Method** **Chapter 3.2 · return and void**

Chapter 3.2 · Pass by Value

CHECK QUESTION

▶ **After the definition static void inc(int x) { x++; } , if int a = 5; inc(a); is run, what is a?**

EXAMPLE (JAVA)

```
static int square(int x) { return x * x; }
static void greet(String name) { System.out.println("Hello " + name); }

// inside main:
int r = square(5) + square(2);   // 29
greet("Ann");                    // Hello Ann
```

[ ] Arrays and strings: a first look

index from 0 · length · charAt

An array holds a fixed number of values of a single type; the values are numbered with indices from 0 to length − 1. A String is a sequence of characters whose positions are also numbered from 0. If the program tries to read outside these bounds, it stops with an exception.

WHY IT MATTERS

Grade lists, matrices, text processing, and the collection classes all rest on access by index.

WHERE IT IS USED **Chapter 3.3 · Declaring Arrays** **Chapter 3.3 · Access and length**

Chapter 3.4 · Indexing **Chapter 3.3 · Index Out of Bounds**

CHECK QUESTION

▶ **int[] a = {4, 8, 15}; What happens if a[3] is read?**

EXAMPLE (JAVA)

```
int[] a = {4, 8, 15};
System.out.println(a.length + " " + a[0] + " " + a[a.length - 1]);   // 3 4 15
String s = "Java";
System.out.println(s.length() + " " + s.charAt(1));                  // 4 a
```

Tools and setup

Installing Java, compiling and running programs, and reading what the tools report.

>_ Command-line basics

cd · dir / ls · mkdir · working directory

On the command line (Command Prompt or PowerShell on Windows, Terminal on macOS and Linux), programs are run by typing their names. The prompt shows the current working directory; cd changes this directory, dir (ls on macOS/Linux) lists files, and mkdir creates a folder. The Tab key completes file names, and the up-arrow key brings back earlier commands.

WHY IT MATTERS

javac and java are command-line programs; running them by hand shows what an IDE's Run button does behind the scenes.

WHERE IT IS USED **Chapter 2.2 · Compiling and Running from the Command Line**

Chapter 2.2 · Programming Tools

CHECK QUESTION

▶ **The prompt looks like C:\Users\ann\course>. What is the working directory after the command cd .. ?**

EXAMPLE (WINDOWS COMMAND LINE)

```
C:\Users\ann> mkdir course        creates a folder
C:\Users\ann> cd course           enters the folder
C:\Users\ann\course> dir          lists the files (ls on macOS/Linux)
C:\Users\ann\course> cd ..        goes up one level
C:\Users\ann>
```

JDK

Installing the JDK

JDK · LTS release · PATH · java -version

The Java Development Kit (JDK) contains the javac compiler, the java launcher together with the JVM, and the standard library. A long-term support (LTS) release such as JDK 17 or 21 (for example from Adoptium or Oracle) is a good choice. For the commands to work from any directory, the JDK's bin folder must be added to the PATH; the installer or the user does this. The commands java -version and javac -version verify the installation.

WHY IT MATTERS

Without a working JDK, neither the command-line examples nor an IDE can compile Java code.

WHERE IT IS USED

Chapter 2.2 · The Java Platform **Chapter 2.2 · Programming Tools**

CHECK QUESTION

▶ **The command line answers 'javac' is not recognized as an internal or external command. What is a likely cause?**

EXAMPLE (CHECKING THE INSTALLATION)

```
> java -version
openjdk version "21.0.3" 2024-04-16 LTS
OpenJDK Runtime Environment Temurin-21.0.3+9 (build 21.0.3+9-LTS)
OpenJDK 64-Bit Server VM Temurin-21.0.3+9 (build 21.0.3+9-LTS, mixed mode,
sharing)
> javac -version
javac 21.0.3
```

▶

javac and java: compiling and running

Hello.java · Hello.class · main

The command javac Hello.java compiles the source file and, if there are no errors, writes Hello.class. The command java Hello then starts the JVM and runs this class's main method (the name is given without the .class extension). A public class must be saved in a file with exactly the same name, including upper and lower case.

WHY IT MATTERS

Every program in the course has this form: a class with a main method is first compiled and then run.

WHERE IT IS USED

Chapter 2.2 · A Sample .java File **Chapter 2.2 · Compilation**

Chapter 2.2 · Compiling and Running from the Command Line **Chapter 2.2 · Bytecode**

CHECK QUESTION

▶ **A file named Test.java contains the definition public class Hello. What does javac Test.java report?**

EXAMPLE (HELLO.JAVA)

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, Java!");
    }
}

> javac Hello.java      (Hello.class is created)
> java Hello
Hello, Java!
```

IDE

Working with an IDE

IntelliJ IDEA · Eclipse · VS Code · debugger

An integrated development environment (IDE) brings together an editor, a compiler, and a debugger: it underlines errors as you type, completes names, and compiles and runs the program with the Run button. Common choices are IntelliJ IDEA, Eclipse, and Visual Studio Code with Java extensions. The debugger stops at a breakpoint and executes the program line by line while showing the variables.

WHY IT MATTERS

Large programs made of several classes are written in an IDE; with long code, the debugger replaces tracing by hand.

WHERE IT IS USED

Chapter 2.2 · Programming Tools **Chapter 7 · The Debugger**

Chapter 7 · Step Into / Over / Out

CHECK QUESTION

▶ **The debugger has stopped at the line int r = square(5); What is the difference between Step Over and Step Into?**

EXAMPLE (USEFUL IDE SHORTCUTS)

```
IntelliJ IDEA   psvm + Tab   ->  public static void main(String[] args)
IntelliJ IDEA   sout + Tab   ->  System.out.println();
Eclipse         main + Ctrl+Space, sysout + Ctrl+Space
in all IDEs     clicking the margin next to a line sets a breakpoint
```

!

Reading error messages

compile-time error · exception · stack trace

javac reports a compile-time error with the file name, the line number, and a caret (^) placed under the problem; in this case no .class file is produced. A run-time error is an exception: the program compiles but stops while running and prints the type of the exception, a message, and the stack trace of method calls. You usually start with the first message and its line number.

WHY IT MATTERS

Fixing errors quickly depends on reading these messages instead of guessing.

WHERE IT IS USED **Chapter 2.1 · Reading Error Messages**

Chapter 2.1 · What Can Go Wrong? **Chapter 6 · Reading a Stack Trace**

CHECK QUESTION

▶ **Is int x = "5"; a compile-time error or a run-time error?**

EXAMPLE (REAL MESSAGES)

```
Hello.java:3: error: ';' expected
        System.out.println("Hello")
                                   ^
1 error

Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Divide.main(Divide.java:4)
```

doc

Reading API documentation

Javadoc · method signature · return type

The Java standard library is documented in Javadoc pages (docs.oracle.com) organized by package and class. A method's header says what it needs and what it returns: in the header public static int parseInt(String s), the parameter is a String and the result is an int; static shows that the method is called through the class, as in Integer.parseInt("42").

WHY IT MATTERS

Classes such as String, Math, Scanner, ArrayList, and HashMap are used through their documented methods.

WHERE IT IS USED **Chapter 3.2 · The Math Class** **Chapter 3.4 · Basic Methods**

Chapter 3.2 · The Most Common Methods

CHECK QUESTION

▶ **Math.max has separate versions for int and double. What does Math.max(3, 7.5) return?**

EXAMPLE (SIGNATURES AND CALLS)

```
public static int parseInt(String s)      // in the Integer class
public int length()                       // in the String class
public static double sqrt(double a)       // in the Math class

int n = Integer.parseInt("42") + 1;       // 43
int k = "Java".length();                  // 4
double r = Math.sqrt(16);                 // 4.0
```

Getting ready for objects

Two habits of thought that object-oriented programming rests on.

is

Classification: is-a and has-a

category · part · relationship

Everyday objects are divided into categories (every dog is an animal) and are made of parts (every car has an engine). Object-oriented programming models the first relationship with inheritance (is-a) and the second with fields that refer to other objects (has-a). Telling these two relationships apart in a sentence is the first step of class design.

WHY IT MATTERS

Choosing between extends and a field, and building class hierarchies, starts from this distinction.

WHERE IT IS USED **Chapter 4.3 · is-a and has-a Relationships**

Chapter 4.1 · Objects: Fields and Behavior **Chapter 4.1 · Class: The Blueprint of an Object**

CHECK QUESTION

▶ **A student has a student number; a student is a person. Which is the is-a and which is the has-a relationship?**

EXAMPLE (A PREVIEW OF JAVA)

```
class Animal { }
class Dog extends Animal { }      // every Dog is an Animal

class Engine { }
class Car {
    Engine engine = new Engine();  // every Car has an Engine
}
```

▣ Abstraction and decomposition

splitting into parts · hiding detail · reuse

Decomposition divides a big problem into small parts that can be solved separately; abstraction keeps only the important aspects of a part and hides how it works inside. Using the Math.sqrt method without knowing its algorithm is abstraction; writing a program as several small methods is decomposition.

WHY IT MATTERS

Methods, classes, interfaces, and encapsulation are the tools of these two ideas.

WHERE IT IS USED **Chapter 3.2 · Decomposition** **Chapter 3.2 · Abstraction**

Chapter 4.5 · Abstraction **Chapter 1.2 · Building Blocks of Computation**

CHECK QUESTION

▶ **A program reads grades, computes their average, and prints a report. Into which parts can this program be divided?**

EXAMPLE (A PREVIEW OF JAVA)

```
static int[] readGrades() { ... }
static double average(int[] g) { ... }
static void printReport(double avg) { ... }

// main only puts the parts together:
printReport(average(readGrades()));
```

The outputs in the Java examples were checked by compiling the programs with javac --release 8 and running them; outputs marked “for example” vary from computer to computer.

APPENDIX C

Important People in Object-Oriented Programming with Java

JAVA · PEOPLE

The 66 people behind the ideas in the course, on 47 cards: from Euclid's algorithm, Babbage's Analytical Engine, Turing's universal machine, and ENIAC to Simula and Smalltalk, James Gosling's Java and the JVM, information hiding and Design by Contract, the Collections Framework, the java.util.concurrent package, the relational model, JUnit, and design patterns.

Each card gives the years the person lived, their country and institution, their best-known contribution, and a short summary; the links lead to the slides where that contribution appears. The cards can be filtered by chapter, searched, and sorted by year of contribution or by surname.

EU

Euclid (Eukleides)

c. 300 BC · Greek · Alexandria

Euclid's algorithm · Elements, c. 300 BC

A Greek mathematician who worked in Alexandria; very little is known about his life. He is the author of the *Elements*, a 13-book work that organizes geometry and number theory through axioms and proofs. Book VII gives the method for finding the greatest common divisor (GCD) of two numbers. In Euclid's account the smaller number is subtracted from the larger again and again; today's form, written with division and remainder, is a shorthand for the same method: when the remainder is 0, the last divisor is the GCD. It is one of the oldest known algorithms still in use. The relation GCD(a, b) = GCD(b, a mod b) turns into a one-line recursive method, and a class that reduces fractions uses the same method in a helper method.

IN THE COURSE **Chapter 1.2 · Euclid's Algorithm** **Chapter 1.2 · Euclid: Trace**

Chapter 3.5 · Greatest Common Divisor **Chapter 4.1 · Design Example: Rational**

Euclid's algorithm GCD imperative knowledge

KH

Muhammad ibn Musa al-Khwarizmi

c. 780 – c. 850 · Baghdad · House of Wisdom

Calculation with Hindu numerals · c. 825

A mathematician and astronomer who worked at the House of Wisdom in Baghdad. His book on calculation with Hindu numerals, written around 825, explains step by step the decimal positional number system and the methods for the four operations with these numerals; the book helped this system spread through the Middle East and Europe. The 12th-century Latin translation of the work became known as *Algoritmi de numero Indorum*. The word “algorithm” derives from the Latinized form of the author's name, and the word “algebra” comes from his book titled *al-Jabr*.

IN THE COURSE **Chapter 1.3 · What Is an Algorithm?** **Chapter 1.2 · Algorithm**

algorithm decimal number system algebra

BA

Charles Babbage and Ada Lovelace

1791–1871 · 1815–1852 · British · London

Analytical Engine · Note G, 1843

Starting in 1834, Babbage designed the Analytical Engine, a general-purpose machine to be programmed with punched cards; the machine was never completed. In her 1843 notes on the engine, Ada Lovelace gave a step-by-step method for computing Bernoulli numbers (Note G); this method is generally regarded as the first published computer program. Lovelace realized that such a machine could work on symbols other than numbers. The engine's separate “store” and “mill” units and its repeated sequences of operations anticipate today's concepts of memory, processor, and loop.

IN THE COURSE **Chapter 1.2 · Babbage and the Analytical Engine**

Chapter 1.2 · Ada Lovelace: The First Program

Analytical Engine first program loop

AC

Alonzo Church

1903–1995 · American · Princeton University

Lambda calculus · 1936

In the early 1930s he developed the lambda calculus, a model of computation based solely on defining anonymous functions and applying them. In 1936 he used this model to prove an undecidability result independently of Turing; this is why the Church–Turing thesis carries both

names. Functional programming languages derive from his calculus. Lambda expressions such as `x -> x * x`, introduced with Java 8, take their name from this calculus as well.

IN THE COURSE **Chapter 4.5 · Anonymous (Lambda) Functions**

Chapter 2.1 · Same Problem: Imperative vs. Functional

lambda calculus anonymous function functional programming

AT Alan Turing

1912–1954 · British · Cambridge, Manchester

Turing machine · 1936

In his paper “On Computable Numbers” (1936) he defined the Turing machine. He showed that a single universal machine can imitate every machine whose description it is given, and he proved that no algorithm can decide in general whether a program will halt (the halting problem). The universal machine is the idea of treating a program as data; stored-program computers turned this idea into reality. The JVM, a program that runs other programs supplied as bytecode, rests on the same principle. The halting problem is also the reason no Java tool can catch every infinite loop.

IN THE COURSE **Chapter 1.2 · Universal Turing Machine** **Chapter 1.2 · Computability**

Chapter 1.2 · Limits in Java

computability halting problem universal machine

FY Ronald A. Fisher and Frank Yates

1890–1962 · 1902–1994 · British · Rothamsted Experimental Station

Fisher–Yates shuffle · 1938

Two statisticians who worked together at Rothamsted on the design of experiments. In their 1938 book *Statistical Tables for Biological, Agricultural and Medical Research* they gave a pencil-and-paper way of shuffling a sequence at random: pick one of the items not yet chosen at random, cross it off the list, and write it at the end of the new sequence. The method produces each of the n! orderings of n items with equal probability. Richard Durstenfeld published the in-place swapping form for computers in 1964; Donald Knuth's *The Art of Computer Programming* popularized it, which is why it is also called the Knuth shuffle. `Collections.shuffle` uses this form: it moves from the end of the list toward the front and swaps the element at each position with one chosen at random from the elements from the start up to that position. The work takes O(n) time.

IN THE COURSE **Chapter 5 · Shuffling**

shuffling Collections.shuffle random permutation

AB

John V. Atanasoff and Clifford Berry

1903–1995 · 1918–1963 · American · Iowa State College

Atanasoff–Berry Computer (ABC) · 1942

The physicist Atanasoff designed an electronic calculating machine in 1937 to solve systems of linear equations; with his graduate student Berry he built it at Iowa State College between 1939 and 1942. The ABC kept numbers in binary, carried out operations electronically with more than 300 vacuum tubes, and stored intermediate results in capacitors on rotating drums, a memory that was refreshed regularly. It was not programmable and did not keep its program in memory: it was a fixed-program machine built for a single job. In 1973 a US court invalidated the ENIAC patent and ruled that the idea of the electronic digital computer had been taken from Atanasoff's work.

IN THE COURSE **Chapter 1.2 · ABC and Colossus** **Chapter 1.2 · Fixed-Program Computer**

ABC binary arithmetic vacuum tube

TF

Tommy Flowers

1905–1998 · British · Post Office Research Station, Dollis Hill

Colossus · 1944

An engineer who worked on telephone exchanges at the British Post Office. He designed Colossus to break messages sent by the German army with the Lorenz cipher machine, and he and his team built it at Dollis Hill. The Mark 1, with 1500 vacuum tubes, was delivered to Bletchley Park in January 1944 and began operating in February; the Mark 2, with 2400 tubes, followed it. Colossus is one of the first programmable electronic digital computers, but its program was not loaded into memory: it was set up with switches and plugboard cables. The work remained a state secret for many years, so Flowers could not talk about what he had done.

IN THE COURSE **Chapter 1.2 · ABC and Colossus**

Colossus Bletchley Park vacuum tube

ME

John Mauchly and J. Presper Eckert

1907–1980 · 1919–1995 · American · University of Pennsylvania, Moore School

ENIAC · 1946

The physicist Mauchly and the engineer Eckert designed and built ENIAC at the University of Pennsylvania's Moore School of Electrical Engineering. The machine, with about 18,000 vacuum tubes, ran its first job in December 1945 and was presented to the public in February 1946. ENIAC was the first large general-purpose electronic computer, but its program was set up with cables and switches; for a new job the machine had to be rewired. This difficulty led to the EDVAC design, which puts the program into memory like data; von Neumann described this design, on which the same team was working, in his 1945 report. In 1946 the two founded their own computer company.

IN THE COURSE **Chapter 1.2 · ENIAC and EDVAC** **Chapter 1.2 · Fixed-Program Computer**

ENIAC EDVAC electronic computer

GN Herman Goldstine and John von Neumann

1913–2004 · 1903–1957 · American · Hungarian-born American · Princeton IAS

Flow diagrams · stored-program design

In the report series “Planning and Coding of Problems for an Electronic Computing Instrument” (1947–48) they planned programs with flow diagrams before writing any code; these diagrams are the forerunners of today's flowcharts. Von Neumann's 1945 “First Draft of a Report on the EDVAC” described a computer that keeps instructions and data in the same memory. Almost all computers, including the computer that runs the JVM, follow this design.

IN THE COURSE **Chapter 1.4 · Algorithms and Flowcharts**

Chapter 1.1 · von Neumann Architecture **Chapter 1.1 · The Stored-Program Concept**

flow diagram stored program von Neumann architecture

GH

Grace Hopper

1906–1992 · American · US Navy, Remington Rand

A-0 compiler · 1952

In 1952 she wrote A-0, one of the first programs to translate symbolic code into machine code, and she argued that programs should be written with words resembling English; her FLOW-MATIC influenced COBOL (1959). The idea of having a program do the translation lives on every time `javac` runs. The moth taped into the Harvard Mark II logbook in 1947 with a note about a "bug" is associated with her team and helped popularize the word; engineers had used the word before that.

IN THE COURSE **Chapter 2.2 · Compilation**

Chapter 2.1 · Compiled and Interpreted Languages **Chapter 7 · Error, Defect, Failure**

compiler high-level language bug

BN

John Backus and Peter Naur

1924–2007 · 1928–2016 · American · Danish · IBM, Copenhagen

FORTRAN · Backus–Naur Form

Backus led the IBM team that produced FORTRAN in 1957; FORTRAN was the first widely used high-level language, with an optimizing compiler that improved the code it generated. In 1959 he proposed a formal notation for syntax; Naur adopted and refined this notation in the ALGOL 60 report (1960), and it became known as Backus–Naur Form. The Java Language Specification defines Java's grammar with a similar notation, and a message such as `';' expected` from `javac` reports that a rule of this grammar has been violated. Both received the Turing Award (1977, 2005).

IN THE COURSE **Chapter 1.2 · Early Programs and Fortran**

Chapter 2.2 · Basic Syntax Rules **Chapter 2.1 · Basic Elements and Syntax**

FORTRAN BNF grammar syntax

JM

John McCarthy

1927–2011 · American · MIT, Stanford University

Lisp · garbage collection

In 1958 he created Lisp at MIT; in his 1960 paper on the language he described garbage collection: memory that the program can no longer reach is reclaimed automatically. Java relies on the same mechanism. When no reference points to an object any longer, the JVM's garbage collector frees it; Java has no operator for deleting an object by hand. He also coined the term “artificial intelligence”; he received the Turing Award in 1971.

IN THE COURSE **Chapter 4.1 · Reference Assignment: Step by Step**

Chapter 3.3 · Where Do Arrays Live in Memory?

garbage collection Lisp references

ED Edsger W. Dijkstra

1930–2002 · Dutch · Eindhoven; Austin, USA

Semaphores · structured programming

In 1965 he introduced the semaphore with its P and V operations and posed the dining philosophers problem, the classic example of deadlock; `java.util.concurrent.Semaphore` offers the same idea through the `acquire()` and `release()` methods. In his 1968 letter “Go To Statement Considered Harmful” he argued for structured control flow; Java reserves `goto` as a keyword but gives it no meaning. He also stressed that testing can show the presence of bugs but never their absence. 1972 Turing Award.

IN THE COURSE **Chapter 10 · Semaphore** **Chapter 10 · Deadlock**

Chapter 2.2 · Keywords **Chapter 7 · Why Test?**

semaphore deadlock goto testing

TH C. A. R. (Tony) Hoare

1934–2026 · British · Oxford, Microsoft Research

Null reference · monitors

In 1965, while designing ALGOL W, he added the null reference to the language; in a 2009 talk he called this decision his “billion-dollar mistake.” Every `NullPointerException` goes back to this decision. In his 1974 paper “Monitors: An Operating System Structuring Concept” he defined the monitor: a shared object whose procedures run one at a time and whose callers can wait for certain conditions. Java gives every object this kind of lock through `synchronized`, `wait()`, and `notify()`. He also invented quicksort; he received the Turing Award in 1980.

IN THE COURSE **Chapter 4.1 · null and NullPointerException**

Chapter 6 · NullPointerException **Chapter 10 · synchronized and Monitors**

Chapter 10 · wait() and notify()

null monitor synchronized

BJ Corrado Böhm and Giuseppe Jacopini

1923–2017 (Böhm) · Italian · Rome

Structured program theorem · 1966

In their 1966 paper “Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules” they showed that every program written as a flow diagram can be rewritten using only three structures: sequence, selection, and repetition. In Java the counterparts of these structures are statements written one after another, the `if` / `switch` statements, and the `while` / `for` statements; no jump statement is needed to express an algorithm. This result became the theoretical foundation of structured programming.

IN THE COURSE **Chapter 1.4 · Control Structures** **Chapter 3.1 · Control Structures**

Chapter 1.2 · Kinds of Algorithmic Operations

sequence selection repetition structured programming

DN Ole-Johan Dahl and Kristen Nygaard

1931–2002 · 1926–2002 · Norwegian · Norwegian Computing Center, Oslo

Simula 67 · classes and inheritance

They designed Simula starting from simulation problems. Simula 67 introduced classes, objects created from classes, subclasses that inherit from a class, and virtual procedures that a subclass can redefine. These are the basic ideas of object-oriented programming: Java's `class`, `new`, `extends`, and overridable methods come from here, and C++ began by adding Simula's classes to C. They shared the 2001 Turing Award.

IN THE COURSE **Chapter 4.1 · Class: The Blueprint of an Object**

Chapter 4.3 · What Is Inheritance? **Chapter 4.4 · Dynamic Binding**

Simula class inheritance virtual method

EC Edgar F. Codd

1923–2003 · British · IBM San Jose

Relational model · 1970

In his 1970 paper “A Relational Model of Data for Large Shared Data Banks” he proposed storing data as relations, that is, tables made of rows, and reaching data through values rather than by following pointers in the storage structure. Every table created and queried through JDBC is a relation in his sense. A SQL query says which rows are wanted, not how they are stored or how to find them; this idea rests on Codd's principle of data independence. 1981 Turing Award.

IN THE COURSE **Chapter 11 · The Relational Model** **Chapter 11 · Files or a Database?**

relational model table data independence

NW

Niklaus Wirth

1934–2024 · Swiss · ETH Zurich

Pascal · P-code

He designed Pascal (1970) to teach structured programming. The Pascal-P compilers of the 1970s produced P-code for a simple virtual stack machine, so porting Pascal to a new computer only required writing a small interpreter. Java's bytecode and the JVM follow the same path to run a single compiled program on many platforms. His book Algorithms + Data Structures = Programs (1976) sums up his view of programming. 1984 Turing Award.

IN THE COURSE **Chapter 2.2 · Bytecode** **Chapter 2.2 · Platform Independence**

Chapter 2.2 · Compilation

Pascal P-code virtual machine portability

AK

Alan Kay

b. 1940 · American · Xerox PARC

Smalltalk · object-oriented programming

In the 1970s he led the group at Xerox PARC that developed Smalltalk. In Smalltalk everything is an object and all computation happens by sending messages to objects. The term “object-oriented programming” is credited to him. A method call in Java such as `car.accelerate(10)` reflects the same thought: a message is sent to the object, and the object decides how to respond using its own data and behavior. 2003 Turing Award.

IN THE COURSE **Chapter 4.1 · Object-Oriented Programming**

Chapter 4.1 · Objects: Fields and Behavior **Chapter 2.1 · Logic and Object-Oriented**

Smalltalk message passing object

DP

David Parnas

b. 1941 · American-born Canadian · Carnegie Mellon, McMaster University

Information hiding · 1972

In his 1972 paper “On the Criteria To Be Used in Decomposing Systems into Modules” he argued that each module should hide a design decision that is likely to change and expose only an interface to the outside. In Java, `private` fields behind `public` methods apply this principle of information hiding: the internal representation of a class can be replaced with another one without changing the code that uses it.

IN THE COURSE **Chapter 4.2 · Information Hiding**

Chapter 4.2 · Changing the Implementation **Chapter 4.1 · Access Modifiers**

information hiding module encapsulation

RK

Dennis Ritchie and Brian Kernighan

1941–2011 · b. 1942 · American · Canadian · Bell Labs

C · The C Programming Language, 1978

Ritchie created C at Bell Labs in the early 1970s; the book he wrote with Kernighan (1978) made printing “hello, world” as a first program famous. Java's visible syntax comes from C: curly braces and semicolons, `for` loops, the `switch` that falls through when `break` is forgotten, `++`, the `&&` and `||` operators, and `printf`-style format strings. Ritchie shared the 1983 Turing Award, given for Unix, with Ken Thompson.

IN THE COURSE **Chapter 2.2 · A Sample .java File**

Chapter 3.1 · Forgetting break: Fall-Through **Chapter 3.4 · The printf Method**

C hello, world printf syntax

PB

Per Brinch Hansen

1938–2007 · Danish · Copenhagen; Syracuse University, USA

Monitors · Concurrent Pascal

He proposed the monitor idea in 1972–73, gave it a class notation in his book Operating System Principles (1973), and built the idea into the language Concurrent Pascal (1975). In his article “Java's Insecure Parallelism” (SIGPLAN Notices, 1999) he argued that the `synchronized` keyword is a weak form of these ideas: nothing prevents unsynchronized methods or non-private fields from being used by several threads at the same time. Lost updates on a shared counter are exactly this kind of bug.

IN THE COURSE **Chapter 10 · synchronized and Monitors** **Chapter 10 · Race Conditions**

Chapter 10 · Why Does counter++ Get Lost?

monitor race condition mutual exclusion

CB Donald Chamberlin and Raymond Boyce

living · 1946–1974 · American · IBM San Jose

SEQUEL / SQL · 1974

At IBM they designed SEQUEL (1974), an English-like query language for Codd's relational model; a trademark conflict led to the language being renamed SQL, and SQL became an ANSI standard in 1986. Commands such as `CREATE TABLE`, `INSERT`, and `SELECT … WHERE` are sent from Java with `Statement` or `PreparedStatement`. Boyce, who also gave his name to Boyce–Codd normal form, died in 1974 at the age of 27.

IN THE COURSE **Chapter 11 · Common SQL Commands** **Chapter 11 · SELECT in Detail**

Chapter 11 · PreparedStatement

SQL query language SELECT

TR Trygve Reenskaug

1930–2024 · Norwegian · Xerox PARC; University of Oslo

Model–View–Controller · 1979

In 1978–79, as a visiting researcher at Xerox PARC, he defined the Model–View–Controller (MVC) structure for Smalltalk: the model holds the data, the view displays it, and the controller handles the user's input. Swing uses a modified form of this separation; a `JTextField` is the view, while the `Document` obtained with `getDocument()` is the model that stores the text. MVC is also one of the standard architectural patterns.

IN THE COURSE **Chapter 9 · Fields and Constructor** **Chapter 9 · AWT and Swing**

Chapter 4.5 · Design Patterns

MVC Swing GUI architecture

GY

Jim Gray

1944–2007 · American · IBM, Tandem, Microsoft Research

Database transactions · 1981

He worked out how database transactions are carried out with locking and logging; in his article “The Transaction Concept: Virtues and Limitations” (1981) he defined transactions as atomic, consistent, and durable. In 1983 Theo Härder and Andreas Reuter added isolation and coined the acronym ACID. In JDBC, `setAutoCommit(false)`, `commit()`, and `rollback()` make sure a money transfer happens either completely or not at all. 1998 Turing Award; he was lost at sea in 2007.

IN THE COURSE **Chapter 11 · Transactions and ACID** **Chapter 11 · A Transfer Example**

transaction ACID commit/rollback

BS

Bjarne Stroustrup

b. 1950 · Danish · Bell Labs; Texas A&M, Columbia University

C++ · 1983

In 1979 he began work at Bell Labs on “C with Classes,” which added Simula's classes to C; the language was renamed C++ in 1983. Java took most of C++'s syntax for classes, constructors, and the `public`, `private`, and `protected` access modifiers; it deliberately left out multiple inheritance of classes, operator overloading, and explicit pointers. In Java a class instead extends a single class and can implement several interfaces.

IN THE COURSE **Chapter 4.3 · Multiple Inheritance** **Chapter 4.5 · Multiple Interfaces**

Chapter 2.1 · Popular Programming Languages

C++ multiple inheritance access modifiers

BL

Barbara Liskov

b. 1939 · American · MIT

Abstract data types · substitution principle

In 1974, together with Stephen Zilles, she defined abstract data types, which are specified by their operations rather than their representation, and built them into the language CLU. In her OOPSLA 1987 keynote “Data Abstraction and Hierarchy” she stated the rule now called the Liskov substitution principle: an object of a subtype must be usable anywhere an object of its supertype is expected. She formalized the principle with Jeannette Wing in 1994. Both a correctly built is-a relationship and a polymorphic array that calls the same method on each of its elements rest on this rule. 2008 Turing Award.

IN THE COURSE **Chapter 4.3 · is-a and has-a Relationships**

Chapter 4.3 · Design Principles for Inheritance **Chapter 4.2 · Interface**

abstract data type subtype LSP

BM

Bertrand Meyer

b. 1950 · French · Eiffel Software, ETH Zurich

Design by Contract · Eiffel

In the mid-1980s he built the Eiffel language on the idea of Design by Contract: each routine states a precondition that the caller must satisfy and a postcondition that the routine itself guarantees; at class level a class invariant is also defined. In his book Object-Oriented Software Construction (1988) he also stated the open–closed principle: modules should be open for extension but closed for modification. Because Java has no built-in contracts, the preconditions of public methods are usually checked by throwing `IllegalArgumentException`, and internal assumptions with `assert`.

IN THE COURSE **Chapter 4.2 · Preconditions and Postconditions**

Chapter 4.2 · Contract: A Complete Example **Chapter 6 · IllegalArgumentException**

Chapter 4.4 · Why Polymorphism?

Design by Contract precondition open–closed principle

GB

Grady Booch

b. 1955 · American · Rational Software, IBM

Booch method · UML

His book Object-Oriented Analysis and Design with Applications (1991) introduced the Booch method and its drawing conventions for showing classes and the relationships between them. At Rational Software in 1994 he began merging his method with James Rumbaugh's, and in 1995 Ivar Jacobson joined them. The resulting Unified Modeling Language (UML) was adopted as a

standard by the Object Management Group in 1997. The UML class box made of a name, attributes, and operations, and the arrow with a hollow triangle head that shows inheritance, are the usual way of drawing class hierarchies.

IN THE COURSE **Chapter 4.3 · The Animal Class Hierarchy**

Chapter 4.1 · Example: The Car Class **Chapter 4.5 · Implementing an Interface**

UML class diagram object-oriented design

JR

James Rumbaugh

b. 1947 · American · General Electric R&D, Rational Software

OMT · UML

At General Electric's research center he led the development of the Object Modeling Technique (OMT), described in the book Object-Oriented Modeling and Design (1991). OMT described a system with an object model made of classes and relationships and a dynamic model made of state diagrams. In 1994 he joined Rational to merge OMT with the Booch method in UML. The life cycle of a thread, which passes through states such as NEW, RUNNABLE, BLOCKED, and TERMINATED, is conventionally drawn with UML state diagrams.

IN THE COURSE **Chapter 4.4 · Example Class Hierarchy** **Chapter 10 · Thread States**

Chapter 10 · State Diagram

OMT UML state diagram

IJ

Ivar Jacobson

b. 1939 · Swedish · Ericsson, Objectory, Rational Software

Use cases · OOSE

While working on telephone exchange software at Ericsson, he developed use cases, which describe a system by what its users do with it; he presented them at OOPSLA 1987 and built the OOSE method around these scenarios (book, 1992). His company Objectory joined Rational in 1995, and use case diagrams became part of UML. Starting from what the user needs before choosing classes, and deriving tests from those needs, follows the same line of thought.

IN THE COURSE **Chapter 1.3 · Defining the Problem** **Chapter 4.2 · Class Design**

Chapter 7 · Test Levels

use case requirements UML

TP Ken Thompson and Rob Pike

b. 1943 · b. 1956 · American · Canadian · Bell Labs

UTF-8 · 1992

In September 1992 they designed the UTF-8 encoding of Unicode: in this encoding ASCII characters stay a single byte, while other characters take two to four bytes. A letter such as é takes two bytes in UTF-8; when a UTF-8 file is read with the wrong character set, such letters appear garbled. In Java a `char` is a 16-bit UTF-16 unit; that is why text read from a file is decoded and the charset can be specified explicitly. Thompson is also one of the creators of Unix; the two later designed the Go language together.

IN THE COURSE **Chapter 1.1 · UTF-8** **Chapter 1.1 · Unicode**

Chapter 8 · Specifying the Character Encoding

UTF-8 Unicode character encoding

GF Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

Vlissides 1961–2005 · Zurich · Sydney · Urbana-Champaign · IBM Research

Design Patterns · 1994

The book Design Patterns: Elements of Reusable Object-Oriented Software (1994), whose authors are known as the “Gang of Four,” collected 23 design patterns in creational, structural, and behavioral groups. Many of these patterns can be seen directly in the Java library: wrapping a stream in a `BufferedReader` is the Decorator pattern, `Iterator` is the Iterator pattern, and a `Comparator` is the Strategy pattern. Event listeners follow the Observer pattern; an abstract class that fixes the steps of an algorithm is the Template Method pattern.

IN THE COURSE **Chapter 8 · Wrapper (Decorator) Streams**

Chapter 4.5 · Template Method Trace **Chapter 5 · Iterator**

Chapter 9 · Listener Interface and Adapter **Chapter 4.5 · Design Patterns**

design patterns Decorator Strategy Observer

JG

James Gosling

b. 1955 · Canadian · Sun Microsystems

Java · 1995

In 1991 Sun's Green project began work on a language called Oak, designed by Gosling for consumer devices; the language, renamed Java, was announced in May 1995, and JDK 1.0 came out in January 1996. Gosling wrote the first compiler and virtual machine: `javac` translates source code into bytecode, and the JVM runs that bytecode on every platform. This principle is summed up in the slogan “write once, run anywhere.” With Bill Joy and Guy Steele he wrote the book The Java Language Specification (1996).

IN THE COURSE **Chapter 2.2 · Compilation**

Chapter 2.2 · Compiling and Running from the Command Line

Chapter 2.2 · Java Virtual Machine **Chapter 2.2 · Bytecode Stack: Code Trace**

Java JVM bytecode javac

LY

Tim Lindholm and Frank Yellin

living · American · Sun Microsystems

The Java Virtual Machine Specification · 1996

Their book The Java Virtual Machine Specification (1996) defined the `.class` file format, the bytecode instruction set, and bytecode verification independently of any implementation. Thanks to this specification, a class file compiled once can run on different JVMs such as HotSpot or OpenJ9, and `javap -c` shows the same instructions everywhere. Lindholm later became the specification lead of JSR 133, the overhaul of the Java memory model.

IN THE COURSE **Chapter 2.2 · Bytecode Example** **Chapter 2.2 · Java Virtual Machine**

Chapter 10 · The Java Virtual Machine (JVM)

JVM specification class file bytecode

KB

Kent Beck and Erich Gamma

b. 1961 · b. 1961 · American · Swiss

JUnit · 1997

Beck had written a testing framework for Smalltalk called SUnit. On the flight to the OOPSLA 1997 conference, he and Gamma wrote the first version of JUnit using pair programming and writing the tests first. JUnit made automated unit tests with assertions such as `assertEquals`

an ordinary part of Java development and gave rise to the xUnit family. Beck also introduced Extreme Programming (1999) and described test-driven development in his book Test-Driven Development: By Example (2002); Gamma later led Eclipse's Java development tools.

IN THE COURSE **Chapter 7 · JUnit** **Chapter 7 · First JUnit Test**

Chapter 7 · Test-Driven Development

JUnit unit test TDD

JB Joshua Bloch

b. 1961 · American · Sun Microsystems, Google

Collections Framework · Effective Java

He led the design of the Collections Framework that came with Java 1.2 (1998): the `List`, `Set`, and `Map` interfaces, implementations such as `ArrayList` and `HashMap`, and the `Collections` algorithms. He also designed the `java.math` package and the `assert` statement. His book Effective Java (2001; third edition 2018) collects rules such as overriding `hashCode` whenever `equals` is overridden and favoring composition over inheritance.

IN THE COURSE **Chapter 5 · Three Parts of the Framework**

Chapter 5 · Programming to an Interface **Chapter 4.3 · equals and hashCode**

Chapter 4.2 · Checking Invariants with assert

collections equals/hashCode Effective Java

GJ Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler

living · Sun Microsystems · University of South Australia · Bell Labs

GJ (Generic Java) · 1998

The GJ design they presented at OOPSLA 1998 added type parameters to Java through erasure: the compiler checks a type such as `List<String>`, then erases the type arguments and inserts explicit casts where needed, so the result runs on existing JVMs. GJ was chosen as the basis of JSR 14, and generics entered the language with Java 5 (2004). Erasure is the reason `new T[10]` and `List<int>` cannot be written. Odersky's GJ compiler became the basis of `javac` starting with JDK 1.3; Odersky later created Scala, and Wadler is one of the designers of Haskell.

IN THE COURSE **Chapter 5 · Generic Types and Type Safety** **Chapter 4.5 · Why Generics?**

generics type erasure type safety

CH

Koen Claessen and John Hughes

living · Swedish · British · Chalmers University of Technology

QuickCheck · 2000

The QuickCheck tool (ICFP 2000) introduced property-based testing for Haskell. Instead of listing sample inputs one by one, the programmer writes a property, for example that reversing a list twice gives back the original list; the tool then tests the property on hundreds of random inputs. Later versions shrink an input that breaks the test down to a small counterexample. The idea has been carried to many languages, including Java libraries such as jqwik and junit-quickcheck.

IN THE COURSE **Chapter 7 · QuickCheck** **Chapter 7 · A Property Test Finds a Bug**

property-based testing random testing

RH

D. Richard Hipp

b. 1961 · American · Charlotte, North Carolina

SQLite · 2000

In 2000 he started SQLite: a database engine that keeps the whole database in a single file, needs no server, and is released into the public domain. Its developers describe it as the most widely used database engine. Because no separate server is needed, a JDBC driver and a connection URL such as `jdbc:sqlite:school.db` are enough for a Java program to create tables and run SQL.

IN THE COURSE **Chapter 11 · What Is SQLite?** **Chapter 11 · Server-Based vs Embedded** **Chapter 11 · Connecting to SQLite**

SQLite embedded database JDBC

TP

Tim Peters

living · Python core developer

TimSort · 2002

In 2002 he wrote TimSort for Python's list sorting. The algorithm finds the already sorted stretches in the data (runs), extends short runs with insertion sort, and merges the runs with merge sort. It is stable: the order of items that compare as equal is preserved. It takes O(n log n)

in the worst case and about n comparisons on already sorted data. Java SE 7 switched the sorting of object arrays to TimSort; `Arrays.sort(Object[])`, as well as `List.sort` and `Collections.sort`, which sort lists based on it, use this algorithm. `Arrays.sort(int[])`, which sorts primitive arrays, uses the Dual-Pivot Quicksort of Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch.

IN THE COURSE **Chapter 5 · Sorting Methods**

TimSort stable sort Collections.sort

DL Doug Lea

living · American · SUNY Oswego

java.util.concurrent · 2004

His book Concurrent Programming in Java (1997) collected design patterns for threads, and the concurrency library he distributed freely turned into JSR 166, which he led. The result is the `java.util.concurrent` package of Java 5 (2004): thread pools behind `ExecutorService`, `ReentrantLock`, blocking queues such as `ArrayBlockingQueue`, `CountDownLatch`, `Semaphore`, and the atomic classes. He also added the fork/join framework to Java 7.

IN THE COURSE **Chapter 10 · ExecutorService** **Chapter 10 · The ReentrantLock Class** **Chapter 10 · Ready-Made: ArrayBlockingQueue** **Chapter 10 · Atomic Classes**

java.util.concurrent thread pool locks

WP William (Bill) Pugh

living · American · University of Maryland

Java memory model · FindBugs

Around 2000 he showed that Java's first memory model was flawed; together with Jeremy Manson and Sarita Adve he designed the new model adopted in Java 5 through JSR 133. This model defines what the `volatile` and `synchronized` keywords guarantee about when one thread sees what another thread has written. He is also one of the authors of FindBugs, a static analysis tool for Java bytecode (its successor is SpotBugs); he invented skip lists (1990) as well.

IN THE COURSE **Chapter 10 · volatile Fields** **Chapter 10 · The Visibility Problem** **Chapter 7 · Static Analysis: SpotBugs**

memory model volatile static analysis

JW

Jeannette Wing

b. 1956 · American · Carnegie Mellon, Columbia University

Computational thinking · 2006

In her 2006 article “Computational Thinking,” published in Communications of the ACM, she argued that decomposing problems, abstraction, and algorithmic thinking are fundamental skills not only for programmers but for everyone. Splitting a problem into methods and classes and hiding details behind an interface is the application of the same skills in Java. Together with Barbara Liskov she also gave the formal definition of behavioural subtyping (1994).

IN THE COURSE **Chapter 1.2 · Building Blocks of Computation**

Chapter 1.2 · Logic, Problem Solving, Creativity **Chapter 3.2 · Decomposition**

computational thinking abstraction decomposition

BG

Brian Goetz

living · American · Oracle

Java Concurrency in Practice · Java 8 lambda expressions

He is the lead author of Java Concurrency in Practice (2006; with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea), the standard reference on thread safety and visibility. As Java Language Architect at Oracle, he was the specification lead of JSR 335, which brought lambda expressions and the stream library to Java 8 (2014). He then led Project Amber, the source of records, switch expressions, sealed classes, and pattern matching.

IN THE COURSE **Chapter 10 · Thread Safety** **Chapter 4.5 · Anonymous (Lambda) Functions**

Chapter 4.1 · Records **Chapter 4.5 · sealed, non-sealed, permits**

Chapter 5 · The Stream API (java.util.stream)

lambda streams records thread safety

MR

Mark Reinhold

living · American · Sun Microsystems, Oracle

Module system · six-month releases

As chief architect of the Java Platform Group he was the specification lead of Java SE 7, 8, and 9. He led the Java 9 (2017) module system, that is, Project Jigsaw, which split the JDK into modules and introduced the `module-info.java` file as a level above packages. In 2017 he proposed

releasing a new Java version every six months, starting with Java 10 in March 2018; this is why features such as the new `switch` are tied to version numbers such as Java 14.

IN THE COURSE **Chapter 2.2 · The Java Platform** **Chapter 4.2 · Packages**

Chapter 3.1 · The New switch (Java 14+)

modules release cadence Java SE

Years and contributions are based on original publications and encyclopedic sources. “Living” marks people who are alive and whose birth year is not given in published sources. The years given for Euclid and al-Khwarizmi are approximate.