

COURSE TEXTBOOK

Computer Fundamentals and Introduction to Programming

Sercan Külcü

Giresun University · Department of Computer Engineering

Computer Fundamentals and Introduction to Programming

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/computer-and-introduction-to-programming/`

Slides, study questions and simulators are on that page. The code for the programming chapters is in the `github.com/sercankulcu/computer-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 computers and information technology: hardware, operating systems, the Internet and office applications. It then moves on to programming through algorithms and flowcharts, and covers the basic constructs of programming in Java, from variables to recursion. It is made up of the notes for the Computer Fundamentals and Introduction to Programming course taught in the Department of Computer Engineering at Giresun University, and its chapters follow the order of the course.

The book has three parts. In order, they are **Computers and Information Technology** (Chapters 1–3), **From Algorithm to Program** (Chapters 4–5), and **Programming in Java** (Chapters 6–12).

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 the programming chapters is in the companion repository `github.com/sercankulcu/computer-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 · COMPUTERS AND INFORMATION TECHNOLOGY

1 Computer Fundamentals and Computational Thinking 5

1.1 Basic Concepts 6
1.2 Computer Hardware 65
1.3 Computers 123
1.4 The Internet 179
1.5 Computation and Computational Thinking 248

2 Operating Systems 295

2.1 Software and Operating Systems 296
2.2 The Windows Operating System 357

3 Office Applications 424

3.1 Microsoft Word 425
3.2 Microsoft Excel 492
3.3 Microsoft PowerPoint 558

PART II · FROM ALGORITHM TO PROGRAM

4 Algorithms and Introduction to Programming 621

4.1 Algorithms 622
4.2 Flowcharts 673
4.3 Introduction to Programming 737

5 Getting Started with Java 813

5.1 Programs and programming languages 814
5.2 Source code and tools 816
5.3 Java 818
5.4 The first Java program 822
5.5 Escape sequences and string concatenation 829
5.6 The structure of a Java file 836
5.7 Compiling and running 837
5.8 Bytecode and the JVM 843
5.9 The building blocks of the language 848
5.10 Comments 852
5.11 Errors 856
5.12 Before coding 864
5.13 Summary 866
5.14 Key terms 867
5.15 Test Yourself 867

5.16 Further reading 873
5.17 References 873

PART III · PROGRAMMING IN JAVA

6 Variables, Data Types and Operators 875
6.1 Variables 875
6.2 Data types 885
6.3 Type conversion 900
6.4 Operators 908
6.5 Input and output 927
6.6 Examples 936
6.7 Common mistakes 944
6.8 Summary 946
6.9 Key terms 947
6.10 Try it yourself 947
6.11 Further reading 950
6.12 References 950

7 Conditionals 951
7.1 Control flow 952
7.2 Conditions and if 955
7.3 Multiway decisions 961
7.4 Nested decisions 967
7.5 Decisions with logical operators 972
7.6 The conditional operator ?: 978
7.7 Comparing strings: == and equals 980
7.8 switch 982
7.9 Examples 990
7.10 Summary 1005
7.11 Key terms 1006
7.12 Try it yourself 1007
7.13 Further reading 1009
7.14 References 1009

8 Loops 1010
8.1 What is a loop? 1011
8.2 The for loop 1013
8.3 The while loop 1017
8.4 The do-while loop 1019
8.5 break and continue 1022
8.6 Loop errors and verification 1026
8.7 The digit pattern 1031
8.8 Product and divisor patterns 1037
8.9 Loops that work with input 1044
8.10 Nested loops 1047

8.11 Patterns 1049
8.12 Test design 1055
8.13 Summary 1056
8.14 Key terms 1057
8.15 Try it yourself 1057
8.16 Further reading 1060
8.17 References 1060

9 Methods 1061

9.1 What is a method? 1062
9.2 Method definition and call 1064
9.3 The call stack 1069
9.4 The parts of a method definition 1073
9.5 The return value 1077
9.6 Passing parameters 1084
9.7 Overloading 1089
9.8 Variable-length arguments 1095
9.9 Scope 1099
9.10 Built-in methods: Math 1107
9.11 Method design 1109
9.12 Example methods 1113
9.13 Summary 1128
9.14 Key terms 1129
9.15 Test Yourself 1129
9.16 Further reading 1132
9.17 References 1132

10 Arrays 1133

10.1 The concept of an array 1134
10.2 Declaring and creating arrays 1137
10.3 Accessing elements 1142
10.4 Traversing with loops 1145
10.5 Basic array operations 1149
10.6 Copying, the Arrays class and methods 1158
10.7 Searching and sorting 1165
10.8 Two-dimensional arrays 1175
10.9 Array or list? 1183
10.10 Common mistakes 1184
10.11 Summary 1186
10.12 Key terms 1187
10.13 Test Yourself 1187
10.14 Further reading 1193
10.15 References 1193

11 Strings 1194

11.1 What is a string? 1195

11.2 Immutability 1201
11.3 Indexing 1205
11.4 String methods 1207
11.5 Comparison 1220
11.6 Memory: the string pool 1231
11.7 Text input with Scanner 1234
11.8 Concatenation and StringBuilder 1238
11.9 Applications 1243
11.10 Formatting with printf 1261
11.11 Unicode and text length 1264
11.12 Summary 1266
11.13 Key terms 1267
11.14 Test Yourself 1267
11.15 Further reading 1269
11.16 References 1270

12 Recursion 1271

12.1 What is recursion? 1272
12.2 Factorial and the call stack 1275
12.3 Single-call examples 1279
12.4 Exponentiation 1286
12.5 Divide and conquer 1289
12.6 Recursion with two calls 1293
12.7 The call stack and stack overflow 1298
12.8 Recursion or a loop? 1302
12.9 Fibonacci and memoization 1305
12.10 Tail recursion 1313
12.11 Overflow, edge cases and debugging 1315
12.12 Summary 1318
12.13 Key terms 1319
12.14 Test Yourself 1320
12.15 Further reading 1322
12.16 References 1322

APPENDICES

A Glossary of Terms 1323

B Prerequisites for Computer Fundamentals and Introduction to Programming 1353

C Important People in Computing and Programming 1375

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.

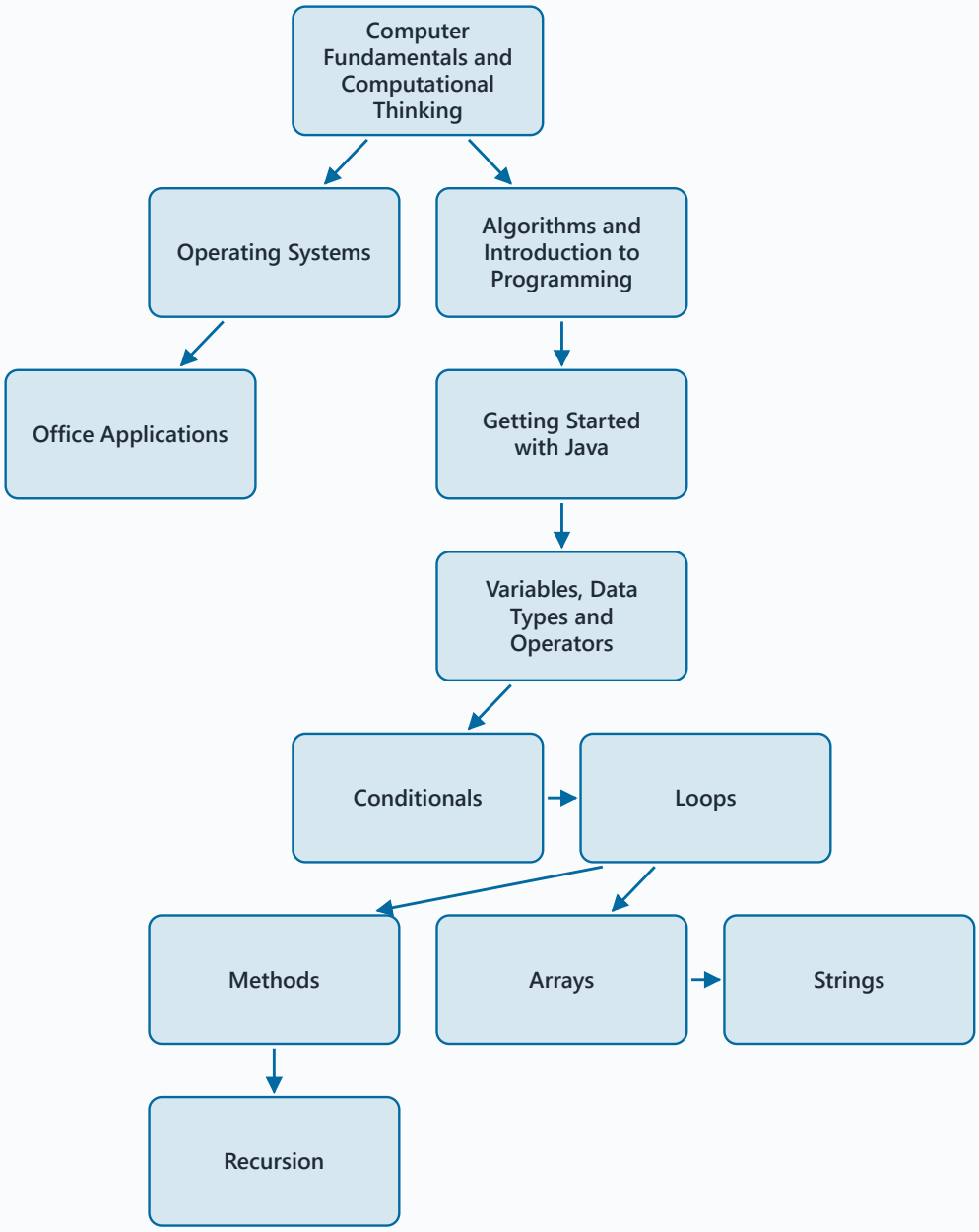


Figure 1: The course map. Arrows show which topic each topic builds on. The course moves from the fundamentals of the computer to algorithms, and from there to programming in Java.

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
Computer Fundamentals and Computational Thinking (1)	—	Data represented as bits, hardware and computational thinking: the ground on which all the concepts of the course rest.
Operating Systems (2)	Computer Fundamentals and Computational Thinking	The software that manages the hardware: booting, processes, memory and the file system.

Topic (chapter)	Builds on	Connection
Office Applications (3)	Operating Systems	Applications that run on the operating system: Word, Excel and PowerPoint.
Algorithms and Introduction to Programming (4)	Computer Fundamentals and Computational Thinking	Computational thinking becomes an algorithm: pseudocode, flowcharts, searching and sorting.
Getting Started with Java (5)	Operating Systems, Algorithms and Introduction to Programming	The algorithm becomes Java code: source code, the compiler, bytecode and the JVM.
Variables, Data Types and Operators (6)	Computer Fundamentals and Computational Thinking, Getting Started with Java	What data looks like in a program: primitive types, overflow, type conversion and operators.
Conditionals (7)	Variables, Data Types and Operators	boolean expressions choose the flow of the program: if, else if, switch.
Loops (8)	Conditionals	Repetition while a condition stays true: for, while, do-while; counters and accumulators.
Methods (9)	Loops	Naming work that repeats: parameters, return values, the call stack.
Arrays (10)	Loops, Methods	Indexed data traversed with loops; the Java code of the searching and sorting from the Algorithms chapter.
Strings (11)	Methods, Arrays	A sequence of characters: String methods, immutability and StringBuilder.
Recursion (12)	Algorithms and Introduction to Programming, Methods	A method that calls itself: the base case, the call stack, Hanoi and binary search.

Table 1: Connections on the map.

The backbone of the map can be read as **computer → algorithm → program → structure → reuse**. The first part introduces the machine and the software that manages it. The Algorithms chapter writes down the steps of problem solving independently of any language; the Java chapters turn these steps into code one after another: first variables and operators, then decisions and loops, and finally methods. Arrays and strings are where loops and methods are used together; recursion makes the method call a tool for shrinking the problem.

PART I

Computers and Information Technology

Chapters 1–3

PART I · CHAPTER 1

Computer Fundamentals and Computational Thinking

A computer is a machine that solves problems by processing data according to precise instructions, and this chapter is taught in five decks. Basic Concepts explains how data is represented in bits and how the basic operating cycle of a computer works, while Computer Hardware surveys the processor, the memory hierarchy, storage, and input and output devices. Computers turns to architecture: two's complement, floating-point numbers, character encodings, logic gates and the fetch–decode–execute cycle. The Internet deck covers IP addresses, DNS, HTTP and HTTPS and safe use of the web, and the last deck introduces computation, the Turing machine and computational thinking.

NOTES IN THIS CHAPTER

1.1 Basic Concepts

1.2 Computer Hardware

1.3 Computers

1.4 The Internet

1.5 Computation and Computational Thinking

PART I · CHAPTER 1.1

Basic Concepts

Whatever a computer is used for, it works in the same simple way: it takes in data, processes it, presents the result, and stores it when necessary. Beneath this cycle lies an even simpler fact. Everything inside a computer, whether text, pictures, or music, is kept as a sequence of 0s and 1s. These notes first describe the working cycle and the hardware that takes part in each stage. They then turn to bits and bytes, the binary number system, and conversions between decimal and binary. Step-by-step calculations show how text, images, sound, and video become numbers, why files are compressed, and how memory sizes and network speeds are measured. Logical operations and the parity bit illustrate how a computer makes decisions with data and catches errors. The last two sections cover the kinds of computers and the place of information technology in everyday life.

LEARNING OUTCOMES

How a computer works

Explain the input, processing, output, and storage stages; identify each stage, and the hardware involved, in a given task.

Digital representation

Interpret the role of bits, bytes, and the binary number system; convert among binary, decimal, and hexadecimal numbers; calculate the space taken by text, images, and sound.

Data processing

Show how arithmetic, comparison, and logical operations combine in decisions; explain the strength and the limits of error detection with a parity bit.

Kinds of computers

Compare computers by purpose and capacity; choose a suitable type for a given need and justify the choice.

Information technology

Explain how data becomes information and describe the data life cycle; discuss the role of information technology in everyday life and the responsibilities it brings.

1.1.1 How a computer works

A calculator, a smartphone, a game console, and a bank's large servers look nothing alike from the outside. Their sizes, prices, and jobs differ enormously. Yet all of them work in the same basic way. This section describes that shared design: what a computer is, which stages it passes a task through, and which hardware takes part in each stage.

1.1.1.1 What is a computer?

In everyday speech, "computer" usually brings to mind a desktop case or a laptop. In computer science the concept is broader. The small control board inside a washing machine is a computer, and so are the racks that fill a server room. What they share is not that each is a single-purpose tool for one job, but that each processes data by following the instructions it is given.

Definition 1.1.1

A **computer** is an electronic system that processes the data it receives according to the instructions of a program. The raw values that enter the computer are called **data**; the interpretable result obtained after processing is called **information**. The sequence of instructions that tells the computer, step by step, what to do is called a **program**.

The three parts of the definition should be kept apart. Data is the material to be processed: an exam grade, a temperature reading, a letter typed on the keyboard. The program is the recipe that says what to do with the material: "add these two numbers," "compute the average," "write the result to the screen." The computer is the machine that follows the recipe to the letter. Load a different program into the same machine and it does an entirely different job. This is what sets a computer apart from a calculator: what it does is not built into its hardware but determined by its program.

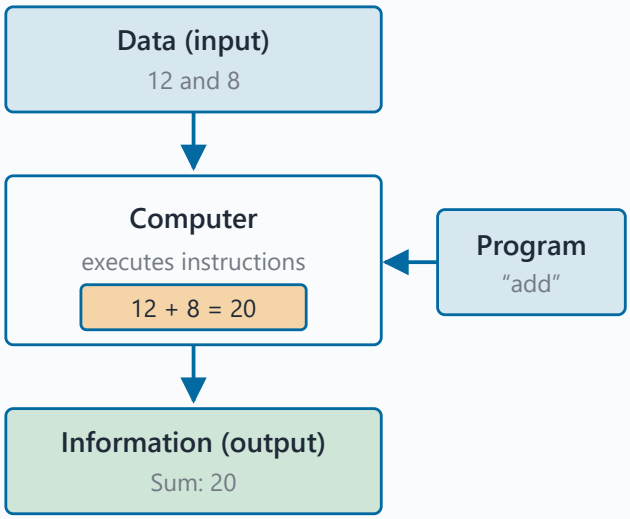


Figure 1.1.1: The simplest model of a computer. Data (12 and 8) enters the computer, the program containing the “add” instruction works on that data, and information (sum: 20) comes out.

Figure 1.1.1 shows this model at the smallest scale. The numbers 12 and 8 mean nothing on their own; we do not know where they came from. The program tells the computer to add them, and the computer produces the result 20. “Sum: 20” is now a result that can be read and interpreted. The distinction between data and information is defined in detail in Definition 1.1.21.

The definition also contains the computer’s basic tasks. A computer *takes in data* from the user or from other systems. It *processes* that data according to the program and produces a meaningful result. It then *presents* the result to the user or *stores* it for later use. These four actions, receiving, processing, presenting, and storing, are the stages of the computer’s working cycle.

Common mistake

Believing that the computer “thinks” or “understands.” A computer carries out the instructions in a program very fast and without error, but it does not know what they are for. When it adds 12 and 8, it does not know whether they are exam grades, prices, or temperatures. It is the human who supplies the meaning and adds context to the data.

1.1.1.2 The basic working cycle

A computer uses the same four stages in every task: **input**, **processing**, **output**, and **storage**. Because of the initial letters, this model is also called the IPOS model. While you type a text, watch a video, or play a game, these four stages repeat millions of times per second. The best way to get to know them is to follow them through one concrete task.

Definition 1.1.2

Input is taking data from the outside world into the computer. **Processing** is calculating, comparing, or transforming the data according to the program's instructions. **Output** is presenting the processed result to the user or to another system. **Storage** is keeping the data and the results so that they can be used again later.

Example 1.1.1

A student information system computes the semester average with the formula *0.4 × midterm + 0.6 × final*. The teacher enters midterm = 70 and final = 90 for one student. The four stages run as follows:

1. **Input.** The teacher types 70 and 90 into the boxes on the form. The keyboard sends every key press to the computer. At this stage the data is only carried into the computer; nothing has been calculated yet.
2. **Processing.** The central processing unit (CPU) applies the formula. First two multiplications are done: *0.4 × 70 = 28* and *0.6 × 90 = 54*. Then the two results are added: *28 + 54 = 82*. These are arithmetic operations.
3. **Output.** The result appears on the screen as "Average: 82." The monitor is the output unit here.
4. **Storage.** The student's grade record is written to the database. So even if the computer is switched off, the record is not lost; when the transcript is prepared at the end of the term, the record is read and processed again.

Result: "Average: 82" appears on the screen, and the record is stored permanently.

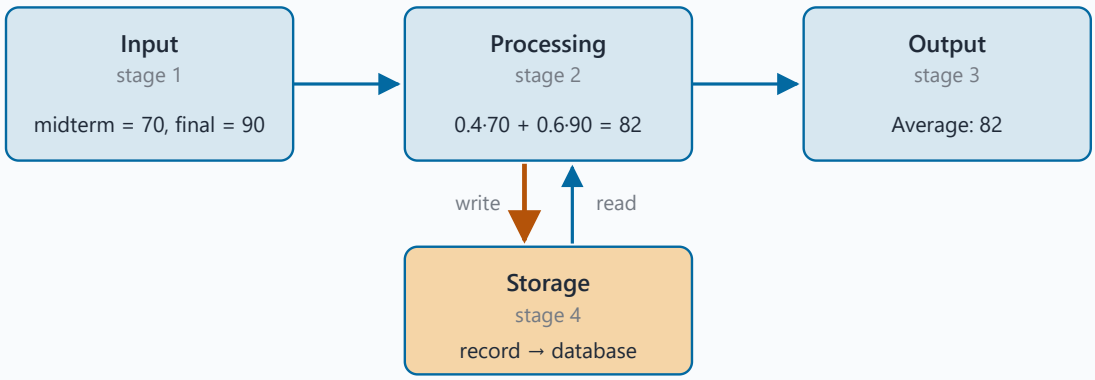


Figure 1.1.2: The four stages of Example 1.1.1. Input (midterm = 70, final = 90) goes to processing, and the result of processing (82) appears at the output. The arrow between storage and processing points both ways: the record is written to the database and read back from it when needed.

Figure 1.1.2 shows how the four stages connect. Input, processing, and output form a one-way line: data enters on the left and leaves on the right as a result. Storage sits beside this line and talks to processing in both directions. Processing can write its result to storage, and if the same data is needed later, it can read it back. In the average example, the transcript calculation at the end of the term is done by reading the grades stored earlier. For that calculation, storage takes the place of input.

The four stages appear under the same names in every task, but they do not carry the same weight in every task. In a calculator, storage is almost absent; the result appears on the screen and is lost at the next calculation. In an archive system, storage is the main job. When applying the model to a task, first look for the answers to four questions: where does the data come from, what calculation is done on it, where does the result go, and what is stored?

Example 1.1.2

When 500 TL is withdrawn from an ATM, the stages break down as follows:

1. **Input:** the card reader reads the card's data; the user enters the PIN and the amount of 500 TL on the keypad.
2. **Processing:** the bank's computer verifies the PIN, compares the balance with the amount to be withdrawn (is balance ≥ 500?), and computes the new balance (balance − 500).
3. **Output:** the cash dispenser hands out the banknotes, the screen shows the new balance, and the printer prints the receipt.
4. **Storage:** the transaction record and the new balance are written to the bank's database.

The processing stage contains two kinds of work: a comparison (is the balance sufficient?) and an arithmetic operation (subtraction). Both are the processor's job.

Common mistake

Mistaking a result shown on the screen for storage. The screen is output; what it displays disappears when the computer is switched off or the screen changes. For a result to count as stored, it must be written to a permanent medium, such as a disk or a database.

1.1.1.3 Input

Input is the computer receiving data from the user, from a sensor, or from another system. The hardware that does this is called an **input unit**. The most familiar input units are the keyboard, the mouse, the microphone, and the touch screen. Sensors are input units too: a thermometer measures temperature, an accelerometer measures the tilt of a phone, and a camera measures light.

Input unit	Physical event sensed	Data sent to the computer
Keyboard	A key is pressed	The key's code (a character or a command)
Mouse	Movement and button presses	Change of position and clicks
Microphone	Fluctuations in air pressure (sound)	A sequence of numbers measured at regular intervals
Sensor	A quantity such as temperature, light, or acceleration	The measured value

Table 1.1.1: Common input units and the data they send to the computer.

All the input units in the table do the same job: they turn a physical event into numbers, or more precisely into bits. There are no keys, sounds, or temperatures inside the computer, only numbers. An input unit's task is to bring an event in the outside world into this form, which the computer can process.

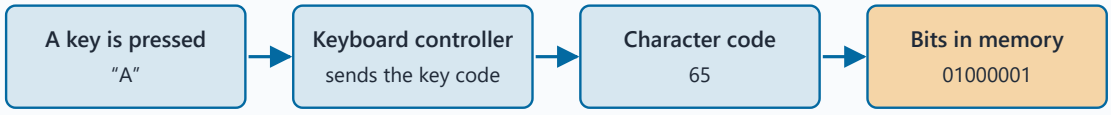


Figure 1.1.3: What happens when the "A" key is pressed. The keyboard controller sends the key code, this code is translated into code 65 for the character "A," and it is stored in memory as the bits 01000001.

Figure 1.1.3 shows this translation for one key press. When a key is pressed, the small controller inside the keyboard sends a code that says which key it was. The operating system translates this code into a character according to the selected keyboard layout. The character code of capital "A" is 65. The number 65 is held in memory as the eight-bit sequence 01000001. Why this sequence corresponds to 65 is worked out step by step in Example 1.1.4.

Common mistake

Counting the touch screen only as an output unit. A touch screen both displays an image (output) and senses the position of the touch (input). It is therefore both an input and an output unit.

1.1.1.4 Processing

At the center of the processing stage is the **central processing unit** (CPU). In everyday speech it is simply called the processor. The processor executes program instructions one after another. Each instruction is a very simple task: add two numbers, read a number from memory, compare two numbers, jump to another instruction depending on the result. Complex programs consist of millions of such small instructions executed in succession.

Definition 1.1.3

The processor has three basic parts. The **control unit** fetches the next instruction, decodes what it asks for, and tells the other parts what to do; it manages the order of the instructions. The **arithmetic logic unit** (ALU) performs calculations such as addition and subtraction, and also comparisons. **Registers** are very small, very fast memory cells inside the processor; they temporarily hold the numbers being worked on.

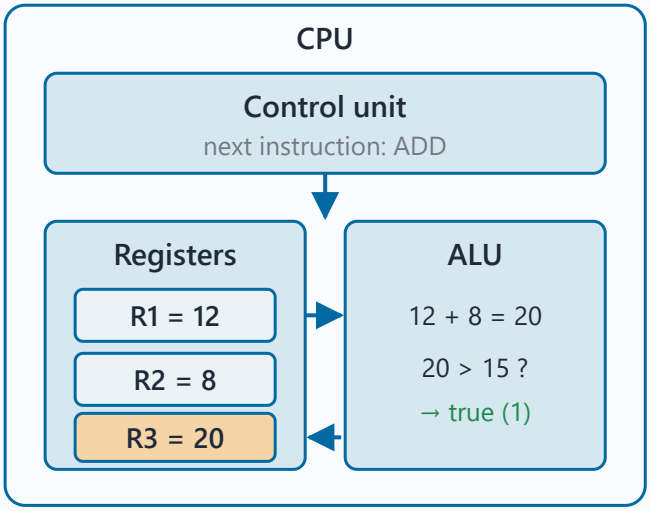


Figure 1.1.4: Inside the processor. The control unit directs the next instruction (ADD). The registers hold R1 = 12 and R2 = 8. The ALU computes the sum (20), writes it to R3, and then performs the comparison 20 > 15; the result is true (1).

Figure 1.1.4 shows the execution of a single addition instruction. The control unit decodes that the next instruction is ADD. The numbers to be added, 12 and 8, were loaded earlier into registers R1 and R2. The ALU reads these two registers, computes *12 + 8 = 20*, and writes the result to register R3. The program's next instruction is a comparison: is 20 greater than 15? The ALU answers this question too and produces 1, which means "true."

The work the ALU does falls into two groups. **Arithmetic operations** calculate with numbers: addition, subtraction, multiplication, division. **Logical operations** compare values and combine conditions: "is 20 > 15?", "are both the score sufficient and the attendance fine?" The result of a logical operation is always "true" or "false." The computer's "decision making," that is, following different paths depending on a situation, is done with these operations. Both kinds of operation are shown in detail on a grade calculation in Figure 1.1.21.

In the processor, both the data and the instructions themselves are represented in binary form, that is, with 0s and 1s. The "ADD" instruction is also a bit sequence in memory; the control unit reads this sequence and understands which job is to be done. Because a program sits in memory just like data, making the computer do a new job takes nothing more than loading a new program into memory.

Common mistake

Measuring a processor's speed only by asking "how many GHz?" The clock speed shows how many basic steps the processor takes per second. The number of steps an instruction takes, the number of cores, and how quickly data arrives from memory also determine speed. Two processors with the same clock speed can run the same program in different times.

1.1.1.5 Output and storage

Output units turn processed information into a form that people can perceive or that another system can use. A monitor presents the result as an image, a printer on paper, and a speaker as sound. This is the reverse of what input units do: bits become light, ink, or sound again. The text "Average: 82" in the average example is the on-screen output of the calculation result.

Storage comes in two kinds, and this distinction matters a great deal in everyday computer use.

Definition 1.1.4

Volatile storage is memory that loses its contents when the power goes off. **RAM** (random access memory), the computer's main memory, is volatile; it is very fast and holds the open programs and documents. **Non-volatile storage** is a medium that keeps data even without power: SSDs, hard disks (HDDs), USB flash drives, and memory cards are of this kind.

When a document is opened, a copy of it is read from non-volatile storage into RAM. Every change made to it is first applied to this copy in RAM. When the "Save" command is given, the current state in RAM is written to non-volatile storage, for example to the SSD. The reason for this order is speed. RAM is much faster than an SSD. If the processor went to the SSD for every letter, the computer would be very slow. The price of speed is safety: everything in RAM is erased when the power goes off.

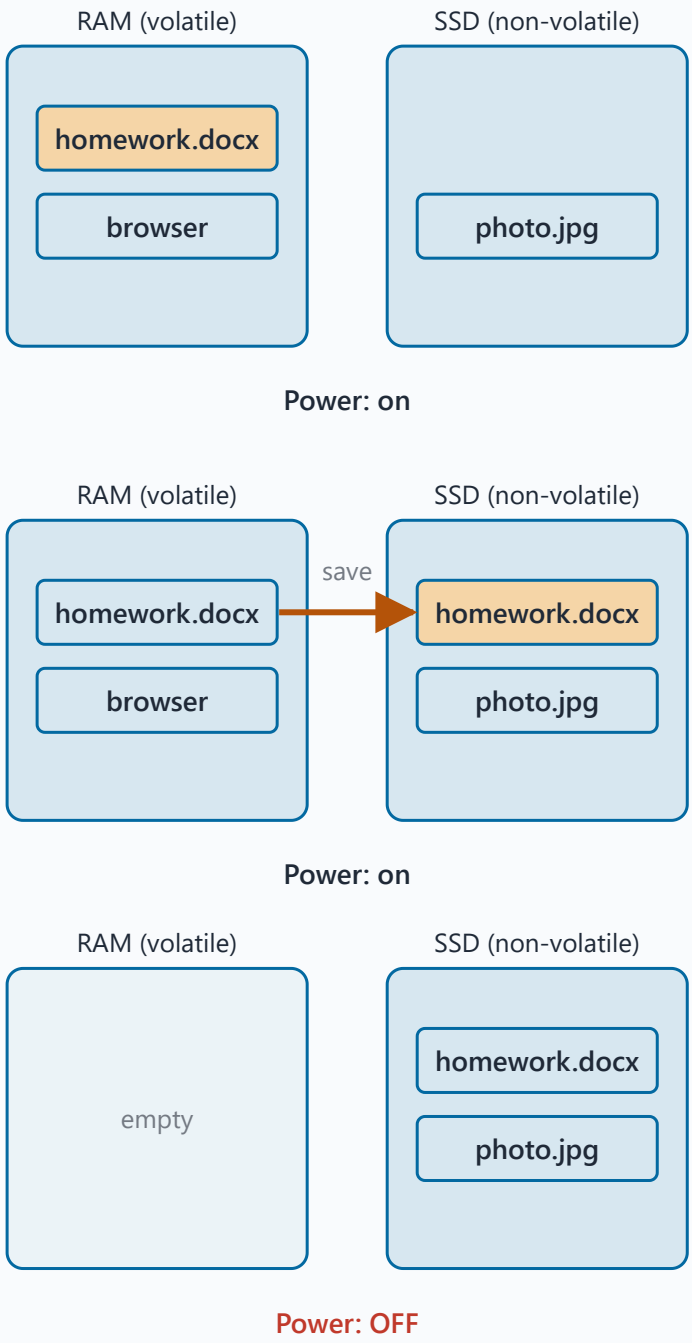


Figure 1.1.5: The division of labor between RAM and the SSD. (a) While the computer is running, the open document (homework.docx) and the browser are in RAM; the SSD holds only photo.jpg. (b) The "Save" command copies the document from RAM to the SSD. (c) When the power goes off, RAM is emptied; homework.docx and photo.jpg on the SSD are preserved.

Figure 1.1.5 shows these three moments in turn. In (a) the document lives only in RAM; if the power went off at that moment, all the work would be lost. In (b) "Save" writes a copy of the document to the SSD. In (c) the power goes off: everything in RAM is erased, but the two files written to the SSD stay where they are. When the computer is turned on again and the document is opened, the last saved state on the SSD is read back into RAM.

Property	RAM (volatile)	SSD, hard disk (non-volatile)
When power goes off	Contents are erased	Contents are preserved
Speed	Very fast	Slow compared with RAM
Typical capacity	8–32 GB	256 GB – a few TB
What it holds	Running programs, open documents	Installed programs, saved files

Table 1.1.2: Comparison of volatile and non-volatile storage.

Example 1.1.3

A student saves a homework document at 2:00 p.m. Between 2:00 and 2:20 she adds two pages to the document but does not save. At 2:20 the power goes off. What is seen when the computer is turned on again and the document is opened?

1. The save at 2:00 wrote the document's state at that moment to the SSD.
2. The changes made between 2:00 and 2:20 were applied only to the copy in RAM.
3. When the power goes off, the contents of RAM are erased; the two added pages are lost.
4. When the document is opened again, the last save on the SSD, that is, the 2:00 state, is read into RAM.

Result: the document opens as it was at 2:00. Some programs have an "autorecovery" feature that writes a copy of the document to disk at regular intervals. If it is on, the loss of the last few minutes is reduced; but that is a separate save made by the program, and RAM itself preserves nothing.

Common mistake

Meaning the disk when saying "my computer's memory is 512 GB." 512 GB is usually the capacity of the SSD; RAM is much smaller (such as 8, 16, or 32 GB). Memory (RAM) and storage (SSD, disk) are separate pieces of hardware and do different jobs.

1.1.2 Digital representation

The previous section said that input units turn every physical event into numbers. This section describes how those numbers are held inside the computer. The answer is surprisingly simple: a computer distinguishes only two states. These two states are called 0 and 1. All numbers, letters, and images are written as long sequences of these two symbols.

1.1.2.1 All data becomes numbers

The data a computer handles is very varied: the letters in a message, the position of the mouse on the screen, the colors of a photograph, the sound of a song. The processor, however, works only with numbers. So for every kind of data there is a rule that turns it into numbers. The details of the rules differ from one kind of data to another, but the idea is always the same: split the data into parts and write each part as a number.

Kind of data	Conversion rule	Example
Text	Each character has a code number.	A → 65 → 01000001
Mouse movement	The position on the screen is written with two coordinates.	(x, y) → (420, 180)
Image	Each pixel is written with red, green, and blue components.	RGB(255, 0, 0): red
Sound	The sound wave is measured at regular intervals.	amplitude → sequence of numbers

Table 1.1.3: How different kinds of data become numbers.

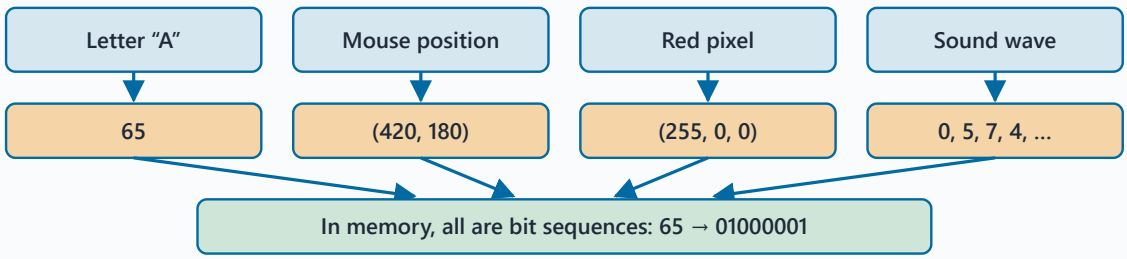


Figure 1.1.6: Four different kinds of data become numbers by four different rules: the letter “A” becomes 65, the mouse position becomes (420, 180), a red pixel becomes (255, 0, 0), and a sound wave becomes the sequence 0, 5, 7, 4, ... In memory all of them sit in the same form, as bit sequences.

Figure 1.1.6 shows these four conversions side by side. The bottom row stresses an important consequence: after conversion, nothing in memory distinguishes a letter from a sound. All of it is bit sequences. Whether a bit sequence is a letter, a number, or a color is known to the program that reads it. This idea is taken up again in Figure 1.1.12.

1.1.2.2 Bits, bytes, and place value

Definition 1.1.5

A **bit**, short for *binary digit*, is the smallest unit of information. A bit takes only one of two values: 0 or 1. A group of eight bits is called a **byte**: 1 byte = 8 bits.

In everyday life we write numbers in the **decimal** system. The decimal system has ten digits (0 to 9), and the value of each position is ten times the value of the position to its right. In the number 352, the 2 is in the ones position, the 5 in the tens position, and the 3 in the hundreds position:

> $352 = 3 \times 100 + 5 \times 10 + 2 \times 1$

The **binary** system applies the same idea with only two digits, 0 and 1. The value of each position is *twice* the value of the position to its right. The rightmost position has value 1; moving left, the values double: 2, 4, 8, 16, 32, 64, 128. These values are the powers of 2: $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, and so on. Positions are counted from the right starting at 0; the digit in position k has value 2^k.

Definition 1.1.6

In a number system, each position has a **place value**. In the decimal system the place values are 1, 10, 100, …; in the binary system they are 1, 2, 4, 8, …. The value of a number is found by multiplying the digit in each position by that position's value and adding the results. To avoid confusion about which system a number is written in, the base is written as a subscript: 101_2 is a binary number, 101_{10} a decimal one.

101_2 and 101_{10} are very different numbers. 101_{10} is one hundred one. 101_2, on the other hand, is $1 \times 4 + 0 \times 2 + 1 \times 1 = 5$. When no base is written, the number is taken to be decimal; but in calculations where binary and decimal numbers appear together, stating the base prevents confusion.

Common mistake

Counting positions from the left. In a binary number the least significant position (the ones position) is always on the *right*, just like the ones position in the decimal system. The 1 in 1000_2 is in position 3 from the right, and its value is $2^3 = 8$, not 1.

1.1.2.3 Converting binary to decimal

To convert a binary number to decimal, write the place value under each bit and add the values of the bits that are 1. Bits that are 0 contribute nothing to the sum. Let us follow the method on the sequence 01000001, which is how the letter "A" on the keyboard appears in memory.

Example 1.1.4

Let us convert 01000001_2 to decimal.

1. **Place values.** Under the eight bits, write the values 1, 2, 4, 8, 16, 32, 64, 128 from right to left.
2. **Bits that are 1.** The second bit from the left (position 6) is 1; its value is 64. The rightmost bit (position 0) is 1; its value is 1. The other six bits are 0.
3. **Sum.** $64 + 1 = 65$.

Result: $01000001_2 = 65_{10}$. In the ASCII table, the number 65 corresponds to capital "A."

Position	7	6	5	4	3	2	1	0
Place value	128	64	32	16	8	4	2	1
Bit	0	1	0	0	0	0	0	1
Contribution	0	64	0	0	0	0	0	1

Table 1.1.4: Place values and contributions to the sum for 01000001_2.

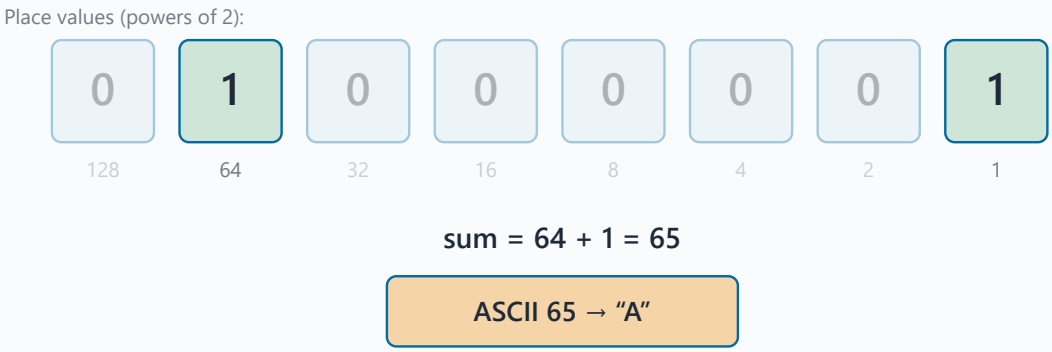


Figure 1.1.7: The bit sequence 01000001 and its place values. Only the bits in the 64 and 1 positions are 1; the sum is *64 + 1 = 65*, and in ASCII 65 is the letter "A."

Figure 1.1.7 is the visual counterpart of the calculation. Once the place values are memorized, converting binary to decimal comes down to adding a few numbers. The values of eight bits (128, 64, 32, 16, 8, 4, 2, 1) form a sequence that everyone who works in computer science ends up memorizing.

Example 1.1.5

What is 11001010_2 in decimal?

1. The bits from left to right are 1, 1, 0, 0, 1, 0, 1, 0; the place values are 128, 64, 32, 16, 8, 4, 2, 1.
2. The values of the bits that are 1: 128, 64, 8, 2.
3. Sum: *128 + 64 = 192*; *192 + 8 = 200*; *200 + 2 = 202*.

Result: $11001010_2 = 202_{10}$. A quick check: the rightmost bit is 0, so the number is even, and 202 is indeed even. If the rightmost bit were 1, the result would have to be odd.

Common mistake

Treating leading zeros as unimportant and shifting the place values. The leading 0 in 01000001 does not change the value of the number; 1000001_2 is also 65. But when the leading 0 is dropped, the number shrinks to seven bits and the leftmost 1 still has value 64, not 128. If place values are always assigned by counting from the *right*, this confusion does not arise.

1.1.2.4 How many values do n bits represent?

A single bit represents two different values: 0 and 1. Two bits can form four different sequences: 00, 01, 10, 11. Three bits give eight sequences: 000 to 111. Each new bit is added by placing a 0 and then a 1 in front of every existing sequence, so the number of sequences doubles at every step.

Theorem 1.1.1 (Number of values of n bits)

Exactly 2^n different bit sequences can be written with n bits. If these sequences are read as unsigned integers, they represent the numbers from 0 to $2^n - 1$.

The reason behind the theorem is a counting argument. Each bit has two choices, and the bits are chosen independently. The first bit has 2 choices and the second again 2, giving $2 \times 2 = 4$ sequences in all. For n bits the choices are multiplied n times: $2 \times 2 \times \ldots \times 2 = 2^n$. The smallest sequence is the one whose bits are all 0, and it represents 0. The largest is the one whose bits are all 1, and its value is $2^n - 1$. The reason is that 0 is counted too: the 2^n values start at 0 and end at $2^n - 1$.

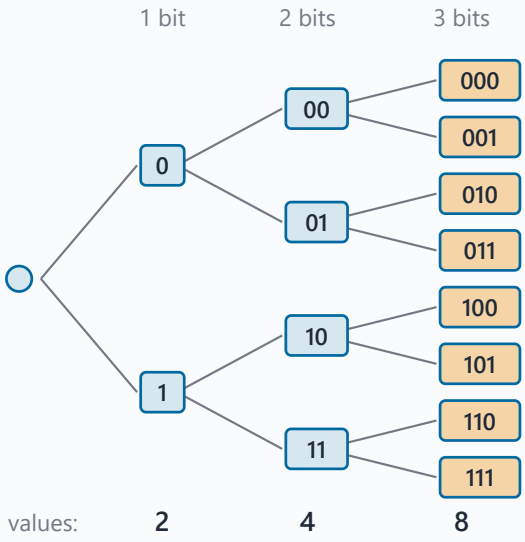


Figure 1.1.8: Each new bit doubles the number of codes. One bit gives 2 codes (0, 1), two bits give 4 (00–11), and three bits give 8 (000–111).

Number of bits	Distinct values	Unsigned range
1 bit	$2^1 = 2$	0–1
2 bits	$2^2 = 4$	0–3
8 bits	$2^8 = 256$	0–255
10 bits	$2^{10} = 1{,}024$	0–1,023
16 bits	$2^{16} = 65{,}536$	0–65,535
32 bits	$2^{32} = 4{,}294{,}967{,}296$	0–4,294,967,295

Table 1.1.5: Number of bits, number of distinct values, and unsigned range.

Table 1.1.5 shows how fast this growth is. 8 bits give 256 values, which is why one byte holds a number between 0 and 255. 10 bits give 1,024 values, and this number comes up again later in the units of memory (1 KiB = 1,024 bytes). With 32 bits, more than four billion different values can be written. Adding one bit doubles the number of values; adding ten bits multiplies it by about a thousand, because $2^{10} = 1{,}024$.

> **Common mistake**
>
> Saying "8 bits count up to 256." 8 bits represent 256 *different* values, but because these values start at 0, the largest is 255. Writing the number 256 takes 9 bits: 100000000_2.

1.1.2.5 How many bits are needed?

The converse of the previous theorem is also asked often: if N different things are each to get a distinct code, what is the minimum number of bits needed? The answer is the smallest n that satisfies $2^n \geq N$. This n is found by starting from 1 bit and adding bits until the number of codes reaches N. The algorithm in the slides expresses this idea in four lines:

```
N = 58   (29 uppercase + 29 lowercase letters)
n = 1; capacity = 2
while capacity < N: n = n + 1; capacity = capacity × 2
capacity ≥ N → n bits suffice
```

In the algorithm, `capacity` holds the number of codes that can be written with `n` bits. At the start there is 1 bit and 2 codes. As long as the capacity stays below `N`, one bit is added and the capacity doubles. The moment the capacity reaches or passes `N`, the loop stops and the current `n` is the answer.

Example 1.1.6

The Turkish alphabet has 29 letters. If uppercase and lowercase letters are coded separately, $N = 58$ different codes are needed. What is the minimum number of bits?

n	capacity = 2^n	capacity < 58?	Decision
1	2	yes	not enough, add a bit
2	4	yes	not enough, add a bit
3	8	yes	not enough, add a bit
4	16	yes	not enough, add a bit
5	32	yes	not enough, add a bit
6	64	no	enough, loop stops

Result: 6 bits are needed. 58 of the 64 codes are used, so $64 - 58 = 6$ codes are left unused. 5 bits are not enough, because 32 codes can cover only 32 of the 58 letters.

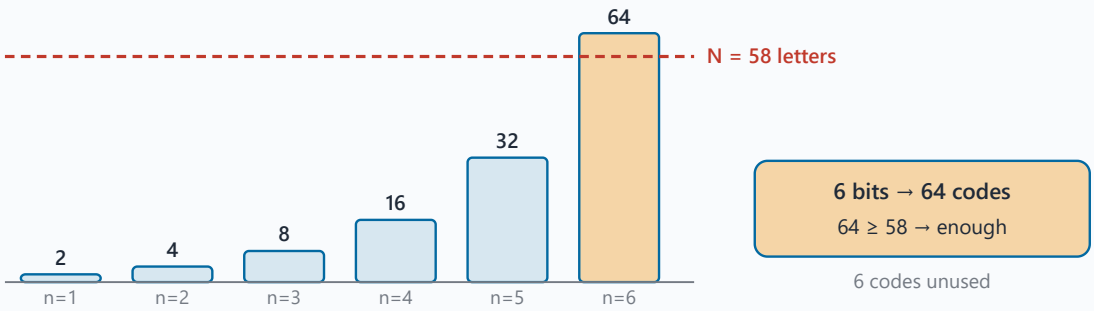


Figure 1.1.9: The number of bits needed for 58 letters. The bars show the number of codes that can be written with $n = 1, \ldots, 6$ bits (2, 4, 8, 16, 32, 64). Only the 6-bit bar passes the 58 line; 6 codes are left unused.

Figure 1.1.9 shows the same result with bars. Each bar is twice the previous one, and the first bar to pass the 58 line is the 6-bit one. A real text does not consist of letters alone: the ten digits, the space, and punctuation marks such as the period, comma, and question mark must also be coded. The **ASCII** (American Standard Code for Information Interchange) table, defined in the 1960s for English text, does this calculation with the English alphabet: 52 uppercase and lowercase letters, 10 digits, punctuation marks, and control characters exceed 64 codes, so ASCII is 7 bits wide (128 codes). The Turkish letters ç, ğ, ı, İ, ö, ş, ü are not in ASCII; for them, extended 8-bit encodings were used first, and today Unicode (UTF-8) is used.

The mathematical counterpart of this calculation is the logarithm: the number of bits needed is $\log_2 N$ rounded up. Since $\log_2 58 \approx 5.86$, 6 bits are needed. For someone who is not comfortable with logarithms, listing the powers of 2 in order gives the same result and is less error-prone.

Common mistake

Confusing the number of values to be coded with the largest value. If the numbers from 0 to 100 are to be coded, the number of values is 101, not 100. $2^6 = 64$ is not enough, and since $2^7 = 128 \geq 101$, 7 bits are needed. The answer would also be 7 bits for the numbers from 0 to 127; for the numbers from 0 to 128, 8 bits would be needed.

1.1.2.6 Converting decimal to binary

The most common method for the reverse conversion is the **repeated-division method**. The number is divided by 2 again and again, and the remainder of each division is noted. The remainder is always 0 or 1: 0 if the number is even, 1 if odd. The division ends when the quotient is 0. Reading the remainders *from the last division to the first*, that is, in reverse order, gives the binary number.

The method can be written as the following algorithm:

```
n ← N
bits ← ""
while n > 0 repeat:
    r ← n mod 2
    bits ← r + bits      (r is added on the left)
    n ← n div 2
print(bits)
```

In the algorithm, `mod` gives the remainder of a division and `div` gives the integer quotient: *45 mod 2 = 1*, *45 div 2 = 22*. Each new remainder is added to the *left* of the bits accumulated so far. So the remainder of the first division ends up at the far right, and the remainder of the last division at the far left.

Example 1.1.7

Let us convert 45 to binary.

Step	Division	Quotient	Remainder	Accumulated bits
1	45 ÷ 2	22	1	1
2	22 ÷ 2	11	0	01
3	11 ÷ 2	5	1	101
4	5 ÷ 2	2	1	1101
5	2 ÷ 2	1	0	01101
6	1 ÷ 2	0	1	101101

The division ends because the quotient is 0 at step 6. Result: $45_{10} = 101101_2$. To check, add the place values: the positions with a 1 in 101101_2 are 32, 8, 4, and 1; *32 + 8 + 4 + 1 = 45*.

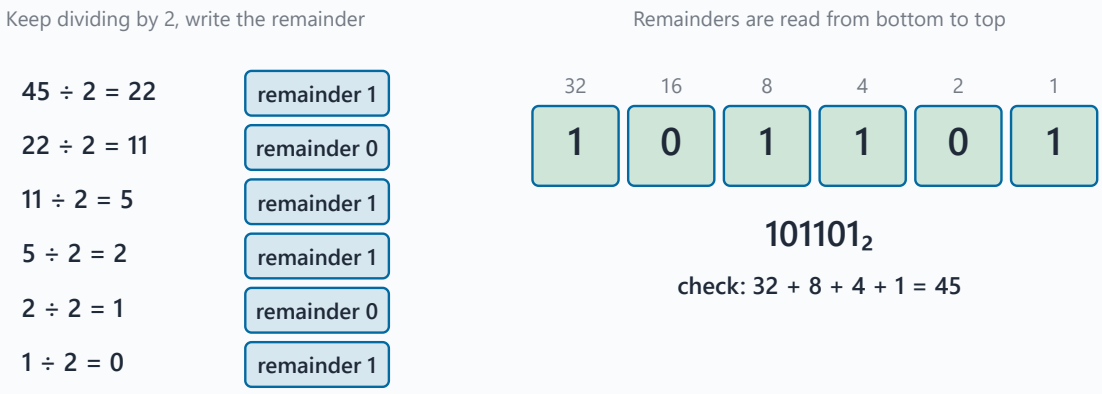


Figure 1.1.10: Converting 45 to binary by repeated division. The remainders of the six divisions on the left fill the place-value boxes on the right from bottom to top: the first remainder goes to the ones position, and the last remainder to the 32s position. The result is 101101_2.

Why the method works can also be explained by a simple observation. The remainder of dividing a number by 2 is the rightmost bit of its binary form: the last bit of even numbers is 0, and of odd numbers it is 1. Dividing the number by 2 and discarding the remainder amounts to deleting the rightmost bit of the binary form. $45 = 101101_2$. Dividing by 2 gives $22 = 10110_2$; the last bit is gone. As the division continues, one more bit is read at each step, and the bits appear one by one from right to left.

Example 1.1.8

Let us convert 156 to binary in two ways.

Method 1: repeated division. 156 ÷ 2 = 78, remainder 0. 78 ÷ 2 = 39, remainder 0. 39 ÷ 2 = 19, remainder 1. 19 ÷ 2 = 9, remainder 1. 9 ÷ 2 = 4, remainder 1. 4 ÷ 2 = 2, remainder 0. 2 ÷ 2 = 1, remainder 0. 1 ÷ 2 = 0, remainder 1. Reading the remainders from last to first gives 10011100_2.

Method 2: subtracting the largest place values. The largest power of 2 that fits into 156 is 128: *156 − 128 = 28*. The largest power that fits into 28 is 16: *28 − 16 = 12*. Then 8: *12 − 8 = 4*. Then 4: *4 − 4 = 0*. The values used are 128, 16, 8, and 4; write 1 in these positions and 0 in the others: $1001\ 1100_2$.

Result: both methods give $156_{10} = 10011100_2$.

For someone who knows the place values well, the second method is usually faster. The first is a mechanical recipe that can be applied without thinking, and it is in a form a program can follow. The two methods always give the same result; one can serve as a check on the other.

Common mistake

Reading the remainders from the first division to the last. For 45 the remainders are 1, 0, 1, 1, 0, 1 in order, and written in that order they give 101101. This wrong reading happens to give the right result for 45, because the sequence is the same backwards. For 156 the remainders are 0, 0, 1, 1, 1, 0, 0, 1. Written in this order, 00111001_2 = 57, which is wrong. The correct reading is from last to first: 10011100_2 = 156. Going back at the end of every conversion and checking with place values catches this mistake at once.

1.1.2.7 Hexadecimal notation

Long bit sequences are hard for people to read and write. In a 32-bit number it is very easy to write a single bit wrong. For this reason bit sequences are usually abbreviated in the **hexadecimal** system. Hexadecimal has 16 digits: 0–9, followed by the letters A, B, C, D, E, F for the values 10–15.

Hexadecimal is convenient because $16 = 2^4$. Four bits represent exactly 16 different values, so each hexadecimal digit corresponds to exactly four bits. Converting a bit sequence to hexadecimal needs no arithmetic: the sequence is split into groups of four starting from the right, and each group is written as a single hexadecimal digit.

Binary	Decimal	Hexadecimal	Binary	Decimal	Hexadecimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

Table 1.1.6: Decimal and hexadecimal equivalents of four-bit groups.

Example 1.1.9

1. **45 → hexadecimal.** 45 = 101101_2. Pad it to eight bits: 0010 1101. 0010 = 2, 1101 = D. The result is $2D_{16}$. Check: $2 \times 16 + 13 = 45$.
2. **255 → hexadecimal.** 255 = 1111 1111_2; both groups are 1111 = F. The result is FF_{16}. The largest value a byte can hold is written FF with two hexadecimal digits.

3. **A color code.** Convert the components of the color with red = 59, green = 130, blue = 246 to hexadecimal. 59 = 0011 1011_2 = $3B_{16}$. 130 = 1000 0010_2 = 82_{16}. 246 = 1111 0110_2 = $F6_{16}$. Written side by side, the components give the color code 3B82F6. Web pages write colors with these six-digit hexadecimal codes.

Hexadecimal notation is not a new kind of data; it is a shorter way of writing the same bit sequence. Memory addresses, color codes, and Unicode character numbers (for example U+0041 for "A") are written in hexadecimal for this reason. By the same logic, groups of three bits can be written as a single digit; this is called **octal** notation: 101 101_2 = 55_8.

Common mistake

Splitting the bits into groups of four starting from the *left*. If 101101_2 is grouped from the left, it becomes 1011 01 and the result is wrong. Grouping is always done from the right; the incomplete group on the left is completed by adding zeros at the front: 10 1101 → 0010 1101 → 2D.

1.1.2.8 Transistors

The answer to why computers use the binary system lies in the hardware. Processors and memories are built from very small electronic parts called **transistors**. A transistor works like a switch that controls the flow of electric current. What turns the switch on or off is not a mechanical lever but an electric voltage.

Definition 1.1.7

A **transistor** is a semiconductor device whose two terminals either pass current or do not, depending on the voltage applied to a third terminal (the gate). When enough voltage is applied to the gate, the transistor **conducts** and current flows; with no voltage it is in **cutoff** and no current flows. In computers these two states are interpreted as 1 and 0.

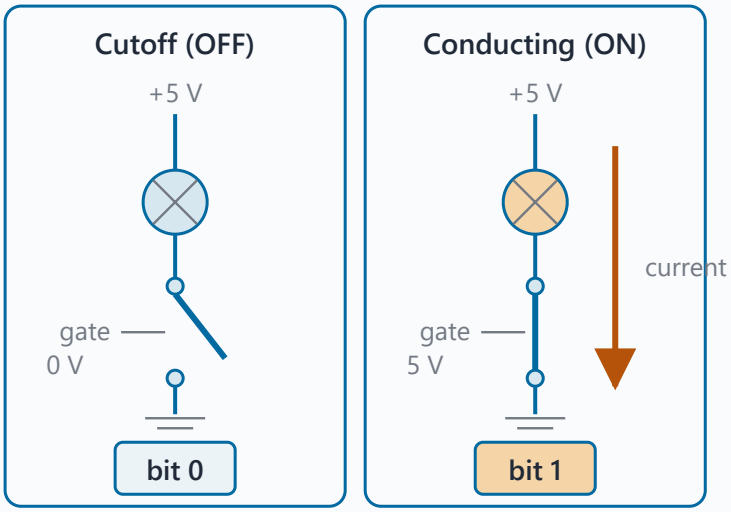


Figure 1.1.11: A transistor works like a switch. On the left the gate voltage is 0 V: the transistor is in cutoff, no current flows, and the lamp is off; this state is bit 0. On the right the gate voltage is 5 V: the transistor is conducting, current flows, and the lamp is lit; this state is bit 1.

Figure 1.1.11 shows these two states with a lamp circuit. It also explains why a computer works in binary and not in decimal. It is easy to tell a switch's two states, open and closed, apart reliably. Telling ten different voltage levels apart is much harder; a little electrical noise can turn a 6 into a 7. In a two-state system, even if the voltage drifts somewhat, the reading is not corrupted, because the gap between "high" and "low" is large.

A single transistor holds a single bit, or does a very simple job on a single bit. The power comes from the numbers. Logic gates are built by combining transistors, adders and memory cells by combining gates, and processors by combining those. A modern phone processor contains tens of billions of transistors. Each can switch on and off billions of times per second. The orderly cooperation of billions of simple switches makes complex jobs possible, such as playing a video or running an AI model.

Common mistake

Thinking of 0 as "nothing" or "no information." 0 is as meaningful a value as 1: it reports that the transistor is in cutoff, that is, a particular state. The six zeros in 01000001 are needed for the number to be 65 at least as much as the two 1s.

1.1.3 Representing data

Bits and bytes are the common alphabet in which a computer writes all its data. This section looks at how different kinds of data are written in this alphabet and how much space they take: numbers and text, images, sound, and video. It then describes two ways of making files smaller, lossless and lossy

compression. The section ends with the units that measure the amount of data and with the difference between bits and bytes in speed and size.

1.1.3.1 Numbers and text

Integers are stored directly as their binary equivalents. The number 13 sits in one byte as 00001101: *8 + 4 + 1 = 13*. Negative numbers and numbers with fractional parts have separate binary encoding rules. For example, a number such as 3.75 is stored by splitting it into three parts: sign, exponent, and significant digits. This format is called a **floating-point number**. The details of these rules are beyond the scope of these notes. What should be remembered is that a rule agreed in advance exists for every kind of number.

Characters, that is, letters, digits, and signs, are converted to numbers by a code table. In the table, every character is given a number. When a computer stores a letter, it actually stores that letter's number; when it writes to the screen, it draws the letter shape that corresponds to the number.

Definition 1.1.8

A **character set** is a table that pairs characters with numeric codes. **ASCII** is a 7-bit table of 128 characters that covers the English alphabet, digits, and punctuation marks. **Unicode** is the universal character standard that covers nearly all the world's writing systems, mathematical symbols, and emoji, and defines more than a million code positions. The first 128 codes of Unicode are the same as ASCII.

The layout of the ASCII table makes some small calculations easy. Uppercase letters start at 65 (A) and are consecutive in alphabetical order: B = 66, C = 67, …, Z = 90. Lowercase letters start at 97 (a). The code of each lowercase letter is exactly 32 more than that of its uppercase form: a = 97 = 65 + 32. The digit characters start at 48: "0" = 48, "1" = 49, …, "9" = 57. There is an important distinction here. The code of the *character* "7" is 55; the *number* 7, however, is stored in binary as 00000111. A "7" typed on the keyboard must be converted before it is used as a number.

Example 1.1.10

How many bytes does the word "Kedi" (Turkish for "cat") take in ASCII, and with which bits is it stored in memory?

Character	Code (decimal)	Bits	Calculation
K	75	01001011	64 + 8 + 2 + 1
e	101	01100101	64 + 32 + 4 + 1
d	100	01100100	64 + 32 + 4
i	105	01101001	64 + 32 + 8 + 1

> Result: four characters, one byte each, 4 bytes (32 bits) in total. The code of K is 10 more than that of A, because K comes 10 letters after A in the alphabet. The code of lowercase e is 32 more than that of uppercase E (69): *69 + 32 = 101*.

ASCII covers only the English alphabet; Turkish letters such as “ğ,” “ş,” and “ı” are not in ASCII. Unicode solves this problem for all languages. In Unicode every character has a **code point**. A code point is written as a hexadecimal number starting with “U+”: “A” = U+0041 (decimal 65), “Ğ” = U+011E (decimal 286), “€” = U+20AC, the Chinese “中” = U+4E2D. How many bytes a code point takes in memory is decided by a separate **encoding** rule. The most common encoding on the web is UTF-8. UTF-8 writes ASCII characters with one byte, Turkish letters with two bytes, and characters such as “€” and “中” with three bytes.

Character	Code point	Decimal	UTF-8 bytes (hexadecimal)	Number of bytes
A	U+0041	65	41	1
Ğ	U+011E	286	C4 9E	2
€	U+20AC	8,364	E2 82 AC	3
中	U+4E2D	20,013	E4 B8 AD	3

Table 1.1.7: Unicode code points of some characters and their byte counts in UTF-8.

This distinction has a practical consequence. “Kedi” is also 4 bytes in UTF-8. “Kağıt” (Turkish for “paper”), however, is 7 bytes even though it has five characters: “K,” “a,” and “t” take one byte each, “ğ” and “ı” two bytes each. The number of characters and the number of bytes differ when non-ASCII letters are used.

The most important consequence of the idea of encoding is this: the meaning of a bit sequence is determined not by the sequence itself but by the rule used when reading it.

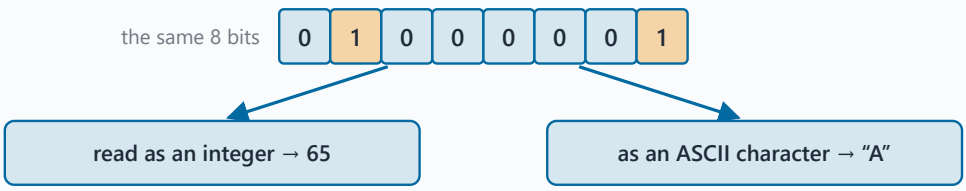


Figure 1.1.12: The same eight bits, 01000001, read by two different rules. Read as an integer, they mean 65; read as an ASCII character, they mean “A.”

Figure 1.1.12 shows this on a single byte. The bits 01000001 in memory are the number 65 to a calculator program, the letter “A” to a text editor, and perhaps a dark gray shade of one pixel to an image program. This is why, when a file is opened with the wrong encoding, meaningless marks such as “Ä±” or “ÅŸ” appear in place of Turkish letters: the bytes are correct, but they were read with the wrong rule.

Common mistake

Saying “a character is always one byte.” This is true only for ASCII. In UTF-8 a character takes between 1 and 4 bytes; Turkish text therefore takes slightly more bytes than it has characters.

1.1.3.2 Images

A photograph seen on a screen consists, up close, of small colored squares. Each square is called a **pixel** (short for “picture element”). A pixel is the smallest part of the image painted a single color. An image is stored as a table of pixels: a certain number of pixels in each row, and one color in each pixel.

Definition 1.1.9

The **resolution** of an image is its number of horizontal and vertical pixels, written “width × height,” such as 1920 × 1080. In the **RGB** color model, the color of each pixel is defined by a mixture of three components: red, green, and blue. Each component is a number between 0 and 255 and takes one byte.

The RGB model is based on how a screen produces light. Each pixel on the screen is really three small light sources side by side: one red, one green, one blue. The value of a component sets the brightness of that light. 0 means the light is completely off, 255 means fully bright. When all three lights are fully bright, the eye sees white: RGB(255, 255, 255). If all three are off, black is seen: RGB(0, 0, 0). If only red is fully bright, the color is pure red: RGB(255, 0, 0). When red and green are lit together, yellow is obtained: RGB(255, 255, 0). If the three components are equal, the color is a shade of gray; the larger the value, the lighter the gray.

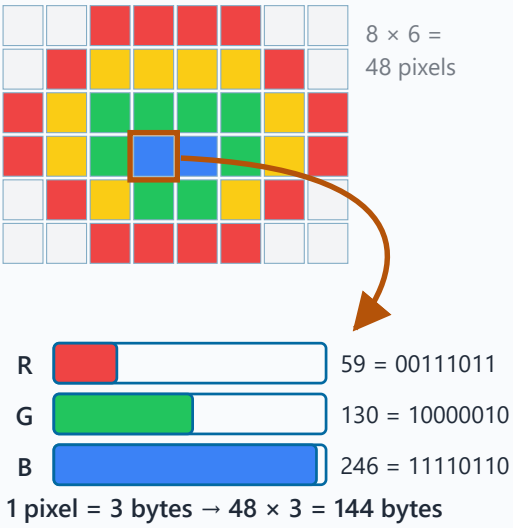


Figure 1.1.13: An image of 8 × 6 = 48 pixels. The components of the selected blue pixel are R = 59 (00111011), G = 130 (10000010), B = 246 (11110110). Since each pixel takes 3 bytes, the image is

48 × 3 = 144 bytes.

Figure 1.1.13 shows this arrangement on a very small image. The image has 8 columns and 6 rows, 48 pixels in all. In the selected blue pixel the blue component has the largest value (246) and the red the smallest (59). Each component is one byte, that is, eight bits. Since a pixel is three bytes, the whole image takes *48 × 3 = 144* bytes. In a real photograph the calculation is the same; only the number of pixels runs into the millions.

Common mistake

Comparing RGB to mixing paint. As paints are mixed, the color gets darker; mixing red, green, and blue paint gives a dark brownish color. On a screen, light is mixed, and as light is added the color gets lighter; the full mixture of the three lights is white.

1.1.3.3 Color depth and image size

The number of bits used to write the color of a pixel is called the **color depth** or **bit depth**. Bit depth determines the number of different colors that can be shown. By Theorem 1.1.1, a pixel of b bits can take 2^b different colors.

Bits / pixel	Distinct colors	Typical use
1	$2^1 = 2$	Black-and-white drawings, fax, document scanning
8	$2^8 = 256$	Grayscale or 256-color palette images (GIF)
24	$2^{24} = 16{,}777{,}216$	Photographs: 8 bits each for R, G, and B
32	16.7 million colors + transparency	Interface graphics, PNG (alpha channel)

Table 1.1.8: Bit depth, number of distinct colors, and typical use.

24 bits give more colors than the human eye can distinguish; this is the standard depth for photographs. At 32 bits the color is still 24 bits; the remaining 8 bits hold transparency information called the **alpha channel**. If alpha is 0 the pixel is fully transparent, and if it is 255 it is fully opaque. It is thanks to this channel that the corners of an app icon show the background color.

Bit depth and resolution together determine the size of an uncompressed image:

> size (bytes) = width × height × bit depth ÷ 8

The ÷ 8 in the formula converts the total in bits to bytes. If the number of bytes per pixel is an integer (such as 24 bits = 3 bytes), the calculation gets even shorter: size = number of pixels × bytes per pixel.

Example 1.1.11

Let us compute the uncompressed size of an 800 × 600 image for three different bit depths. (Decimal units are 1 KB = 1,000 bytes and 1 MB = 1,000,000 bytes; the binary units in parentheses are 1 KiB = 1,024 bytes and 1 MiB = 1,048,576 bytes.)

1. **Number of pixels:** *800 × 600 = 480,000* pixels.
2. **24 bits (3 bytes/pixel):** *480,000 × 3 = 1,440,000* bytes ≈ 1.44 MB (divided by 1,048,576 it is ≈ 1.37 MiB).
3. **8 bits (1 byte/pixel):** *480,000 × 1 = 480,000* bytes = 480 KB (*480,000 ÷ 1,024 ≈ 468.75* KiB).
4. **1 bit:** *480,000 ÷ 8 = 60,000* bytes = 60 KB (*60,000 ÷ 1,024 ≈ 58.6* KiB).

Result: the same image takes ≈ 1.44 MB (1.37 MiB) at 24 bits, 480 KB (468.75 KiB) at 8 bits, and 60 KB (58.6 KiB) at 1 bit. The 24-bit version is exactly 24 times the 1-bit version, because the number of pixels is the same and only the number of bits per pixel has changed.

As color depth increases, color transitions get smoother, but the file gets bigger. In an 8-bit palette image only 256 colors are available, so regions where tones change slowly, such as the sky, show visible steps. At 24 bits the same transition looks continuous to the eye. The effect of resolution is even greater: if width and height are each doubled, the number of pixels, and so the size, quadruples.

Common mistake

Treating bit depth as bytes and forgetting to divide by 8. One gets 800 × 600 × 24 = 11,520,000, and if one says “11.5 million bytes,” the result is eight times too large. 11,520,000 is in bits; to convert to bytes, divide by 8: 1,440,000 bytes.

1.1.3.4 Sound and video

Sound is fluctuation of pressure in the air. A microphone turns this fluctuation into an electric voltage that varies continuously with time. A computer cannot store a continuous wave as it is; it has infinitely many points. Instead, the wave is measured at equal intervals. This process is called **sampling**, and each measurement is a **sample**. Each measurement is also rounded to the nearest of a set of levels and turned into an integer.

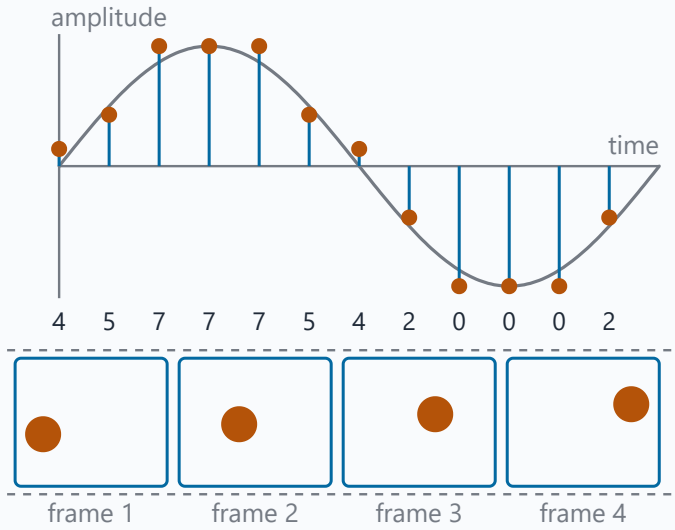


Figure 1.1.14: At the top, a sound wave sampled at 12 points; each sample is rounded to a number between 0 and 7: 4, 5, 7, 7, 7, 5, 4, 2, 0, 0, 0, 2. At the bottom, four consecutive frames of a video; the ball moves a little in each frame.

In the upper part of Figure 1.1.14, the wave is measured at 12 equally spaced points. Samples near the crest of the wave take the value 7, and samples near the trough take 0. The computer stores not the wave itself but these 12 numbers. When the sound is played back, the numbers are turned into a voltage one by one, and the speaker vibrates with that voltage.

Video consists of still images shown one after another. Each image is called a **frame**. When frames are shown fast enough, the eye cannot see them separately and perceives motion. The number of frames shown per second is called **fps** (frames per second). Cinema traditionally uses 24 fps, television and internet videos mostly 30 fps; in games 60 fps and above is common. The sound of a video is stored in a separate channel from the picture frames and is played in sync with the frames during playback.

Example 1.1.12

If a video at 1920 × 1080 resolution, 24-bit color, and 30 fps were stored uncompressed, how much space would one minute take? (Sound is ignored; 1 GiB = 2^{30} = 1,073,741,824 bytes.)

1. **One frame:** *1920 × 1080 = 2,073,600* pixels; × 3 bytes = 6,220,800 bytes.
2. **One second:** 30 frames × 6,220,800 = 186,624,000 bytes, about 187 MB per second (178 MiB).
3. **One minute:** 60 seconds × 186,624,000 = 11,197,440,000 bytes.
4. **Converting to GiB:** 11,197,440,000 ÷ 1,073,741,824 ≈ 10.43 GiB.

Result: one minute of uncompressed Full HD video takes about 10.4 GiB. An ordinary video file with about 100 MB per minute is a hundred times smaller; the difference comes from compression.

1.1.3.5 Digitizing sound

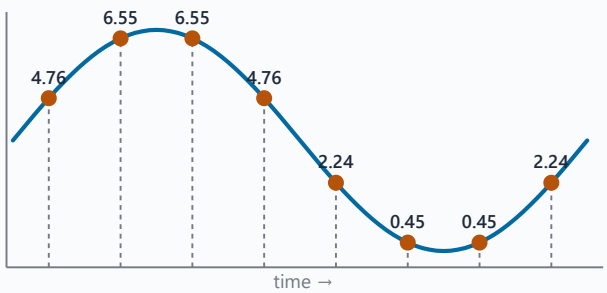
An analog sound signal becomes digital data in three steps: sampling, quantization, and encoding. The algorithm in the slides lists these steps and the size formula as follows:

```
analog signal: continuous wave
sample: measure at equal intervals
quantize: round to the nearest level (3 bits → 0…7)
encode: write each level with 3 bits
size = sampling rate × bit depth × channels × duration
lossy compression (MP3)
```

Definition 1.1.10

Sampling is measuring the signal at equal time intervals; the number of measurements per second is called the **sampling rate** and is measured in hertz (Hz). **Quantization** is rounding each measurement to the nearest of a predetermined finite set of levels; the difference between the measurement and the level is called the **quantization error**. **Encoding** is writing the number of each level as a binary number. The number of bits used to write one sample is called the bit depth.

Let us follow the steps on the small example in the slides. The signal is measured eight times at equal intervals. The measurements have fractional parts: 4.76, 6.55, 6.55, 4.76, 2.24, 0.45, 0.45, 2.24. How the signal changes between two measurements is not recorded; sampling inevitably loses this information.



Sampling

8 measurements at equal intervals

lost between measurements

sampling rate: measurements per second (Hz)

CD: 44,100 Hz

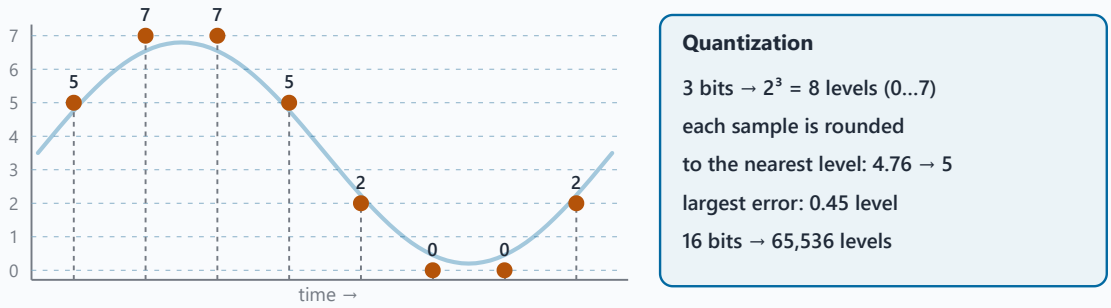


Figure 1.1.15: The first two steps of digitizing sound. Top, sampling: the signal is measured eight times at equal intervals (4.76, 6.55, ...). Bottom, quantization: 3 bits recognize only 8 levels (0...7), and each measurement is rounded to the nearest level; the measurements become 5, 7, 7, 5, 2, 0, 0, 2.

Let us use 3 bits for quantization. By Theorem 1.1.1, 3 bits give $2^3 = 8$ levels: 0, 1, ..., 7. Each measurement is rounded to the nearest integer (Figure 1.1.15, bottom).

Example	Measurement	Level	Quantization error	3-bit code
1	4.76	5	0.24	101
2	6.55	7	0.45	111
3	6.55	7	0.45	111
4	4.76	5	0.24	101
5	2.24	2	0.24	010
6	0.45	0	0.45	000
7	0.45	0	0.45	000
8	2.24	2	0.24	010

Table 1.1.9: Quantizing the eight measurements and encoding them with 3 bits.

Table 1.1.9 gives the results of all three steps together. The quantization error is at most 0.45 of a level. When rounding to the nearest level, the error can never exceed half a level, because every measurement lies between two levels and is at most 0.5 from the nearer one. In the encoding step, each level is written as a 3-bit binary number: 5 → 101, 7 → 111, 2 → 010, 0 → 000. Eight samples take $8 \times 3 = 24$ bits.

This small example grows in two ways for real sound. The first is the sampling rate. Eight measurements per second are not enough to make human speech intelligible; real recordings take thousands of samples per second, and CD quality takes 44,100. The second is the bit depth. Eight levels from 3 bits are like a coarse staircase; a CD uses 16 bits and has $2^{16} = 65{,}536$ levels. On a staircase this fine, the quantization error shrinks below what the ear can hear.

> size (bits) = sampling rate × bit depth × number of channels × duration (s)

Example 1.1.13

How many bytes does one minute of CD-quality sound (44,100 Hz, 16 bits, 2 channels: left and right) take? How much space does the same music take if recorded as 128 kbit/s MP3?

1. **Bits per second:** *44,100 × 16 × 2 = 1,411,200* bit/s.
2. **One minute:** *1,411,200 × 60 = 84,672,000* bits.
3. **Converting to bytes:** *84,672,000 ÷ 8 = 10,584,000* bytes ≈ 10.6 MB (≈ 10.1 MiB if 1 MiB = 1,048,576 bytes is used).
4. **MP3:** 128 kbit/s = 128,000 bit/s. In one minute, *128,000 × 60 = 7,680,000* bits, that is, *7,680,000 ÷ 8 = 960,000* bytes ≈ 0.96 MB (0.92 MiB).
5. **Ratio:** *10,584,000 ÷ 960,000 ≈ 11*.

Result: one minute of CD quality ≈ 10.6 MB (10.1 MiB), while the same minute as a 128 kbit/s MP3 ≈ 0.96 MB; the MP3 is about 11 times smaller.

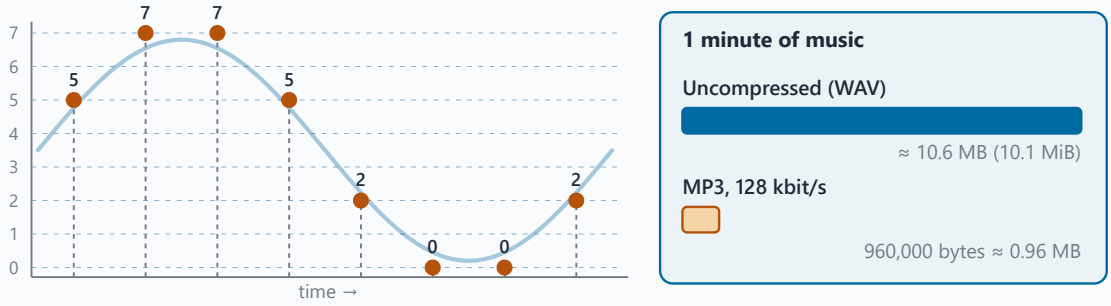


Figure 1.1.16: Sizes of one minute of music uncompressed (WAV, ≈ 10.6 MB = 10.1 MiB) and as MP3 (128 kbit/s, 960,000 bytes ≈ 0.96 MB).

MP3 achieves this reduction with lossy compression: it discards details the ear barely notices, for example very quiet sounds that come right after a loud one. A discarded detail never comes back.

Common mistake

Forgetting the number of channels. In a stereo (two-channel) recording, separate samples are stored for the left and right speakers; the size is twice that of a mono recording. If the factor 2 is left out of the CD calculation, the result is ≈ 5 MB, which is wrong.

1.1.3.6 Lossless and lossy compression

Uncompressed images, sound, and video take a lot of space. For this reason files are usually stored **compressed**. Compression is a way of writing the same data with fewer bits. It comes in two basic kinds, and the difference between them is whether the decompressed file is identical to the original.

Definition 1.1.11

Lossless compression gives back the original data exactly, bit for bit, when decompressed; it works by shortening repetitions in the data. **Lossy compression** permanently discards details that the human eye and ear barely notice; when decompressed, it gives a copy that is close to the original but different from it.

	Lossless	Lossy
On decompression	The original data comes back exactly, bit for bit	A copy close to the original is obtained
Method	Shortens repetitions (RLE, ZIP)	Discards detail the eye and ear barely notice
Example formats	ZIP, PNG, FLAC	JPEG, MP3, MP4 (H.264)
Typical reduction	Usually 2–3 times	10 times or more
Suitable data	Text, programs, spreadsheets, medical images	Photos, music, video

Table 1.1.10: Comparison of lossless and lossy compression.

The nature of the data determines which kind to choose. If even a single bit of a program changes, the program fails or runs wrongly. In a text file, changing a single bit turns one letter into another. In a bank table, 1,000 TL turning into 1,008 TL is unacceptable. This data is compressed only losslessly. In medical images, too, a small detail can matter for diagnosis, so lossless formats are preferred. In photos, music, and video, small changes in the color of a few pixels or in a very quiet sound go unnoticed. This data is compressed lossily and made ten times smaller or more.

Lossy compression has a hidden cost. When a lossy file is opened, edited, and saved again, the compression is applied again, and each time a little more detail is discarded. Editing and saving the same JPEG photo many times produces growing blur and blocky distortions in the image. A discarded detail does not come back. For this reason, original files being worked on are kept in a lossless format, and the lossy version is produced only for sharing.

Common mistake

Thinking that saving a JPEG file as PNG will bring back the lost quality. PNG is lossless, but it preserves only the data it is given. The detail discarded by JPEG compression is nowhere any more; PNG can only store the degraded version without further degradation.

1.1.3.7 Run-length encoding (RLE)

The simplest example of lossless compression is **run-length encoding** (RLE). The idea is this: if the same value repeats many times in a row, instead of writing the value over and over, write "how many times it repeats" and "the value itself." A block of identical consecutive values is called a **run**. For example, the sequence "AAAAABBB" is written (5, A)(3, B).

Definition 1.1.12

RLE is a lossless compression method that reads a sequence from left to right and writes each run as a (length, value) pair. During reading, a counter and the current value are kept. If the incoming item equals the current value, the counter goes up by one. If it differs, the run that has ended is written to the output as (counter, current value), the counter returns to 1, and the new value becomes the current value. When the sequence ends, the last run, still open, is written too.

Example 1.1.14

A 16-pixel row of an image consists of two colors: B (blue) and W (white). With pixel numbers from 0 to 15, the row is BBBBBBWWWWBBBWWW. Each pixel takes one byte; the row is 16 bytes.

Pixel	Color	Event	Counter	Output
0	B	first pixel, current color B	1	—
1–5	B	same color, the counter goes up at each pixel	6	—
6	W	color changed, the run ends	1	(6, B)
7–9	W	same color	4	(6, B)
10	B	color changed	1	(6, B) (4, W)
11–12	B	same color	3	(6, B) (4, W)
13	W	color changed	1	(6, B) (4, W) (3, B)
14–15	W	same color	3	(6, B) (4, W) (3, B)
(last)	—	row ended, the open run is written	—	(6, B) (4, W) (3, B) (3, W)

Result: four pairs are obtained. Each pair is 2 bytes (one byte for the length, one for the color): $4 \times 2 = 8$ bytes. The original row was 16 bytes; RLE has halved it. To check, add the lengths: $6 + 4 + 3 + 3 = 16$ pixels.

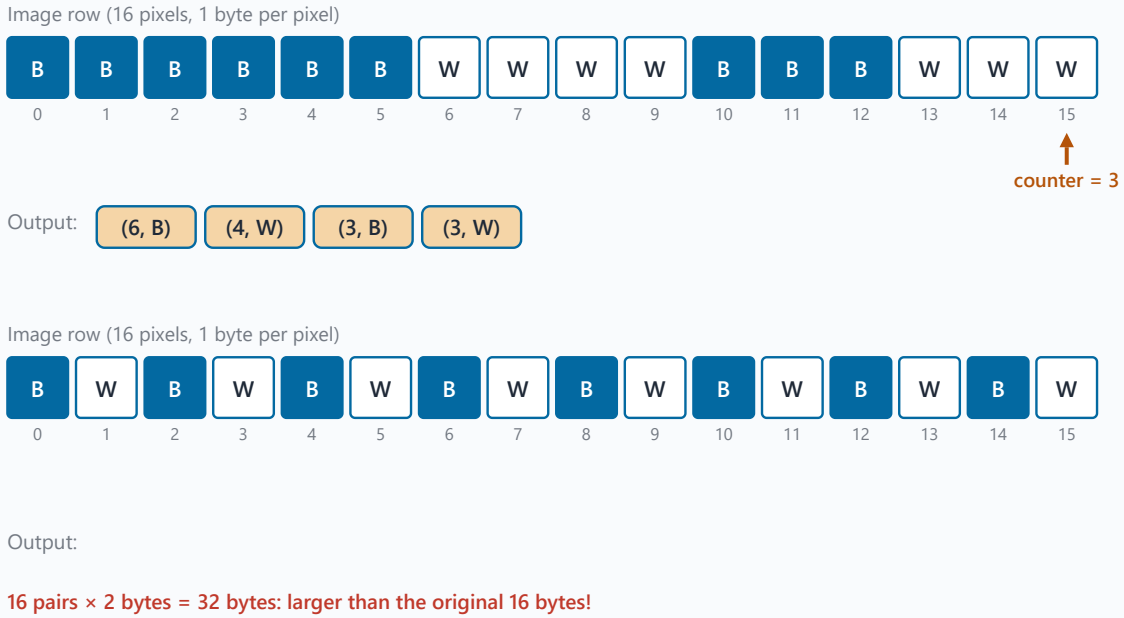


Figure 1.1.17: The good and bad cases of RLE. At the top, the 16-pixel row consists of four runs and shrinks to 8 bytes as (6, B) (4, W) (3, B) (3, W). At the bottom, the color changes at every pixel: every run has length 1, the 16 pairs take 32 bytes, and the output is larger than the original 16 bytes.

Decoding is the reverse of encoding. Each pair is read and the value is written as many times as the length: (6, B) → BBBBBB, (4, W) → WWWW, (3, B) → BBB, (3, W) → WWW. The row obtained is identical to the original row; no information has been lost. This is why RLE is lossless.

The lower part of Figure 1.1.17 shows the limit of the method. If the color changes at every pixel, no run is longer than one. A (1, color) pair is written for each of the 16 pixels: *16 × 2 = 32* bytes. The "compressed" data has doubled in size. RLE works only on data with plenty of long repetitions: black-and-white scanned documents, flat-color drawings, icons. In photographs, neighboring pixels are rarely exactly the same color; so other methods are used for photographs.

Common mistake

Forgetting to write the run at the end of the sequence. RLE writes a run only when the color changes. After the last run the color does not change; the sequence just ends. If this last step is skipped, the output in the example stops at (6, B) (4, W) (3, B), and the last three white pixels are lost; the decoded row is 13 pixels.

1.1.3.8 Units of memory

In the earlier calculations, byte counts reached millions and billions. To write such large numbers comfortably, larger units are used. The units form a ladder that starts at the bit and grows at every step.

Decimal (SI)	In bytes	Binary (IEC)	In bytes
bit (binary digit)	0 or 1		
byte (B)	8 bits		
KB (kilobyte)	10^3 = 1,000	KiB (kibibyte)	2^{10} = 1,024
MB (megabyte)	10^6 = 1,000,000	MiB (mebibyte)	2^{20} = 1,048,576
GB (gigabyte)	10^9 = 1,000,000,000	GiB (gibibyte)	2^{30} = 1,073,741,824
TB (terabyte)	10^{12}	TiB (tebibyte)	$2^{40} \approx 1.1 \times 10^{12}$
PB (petabyte)	10^{15}	PiB (pebibyte)	$2^{50} \approx 1.13 \times 10^{15}$

Table 1.1.11: Units of memory: decimal (SI) and binary (IEC) equivalents.

The first two rows of the table are different: a byte is 8 times a bit. After the byte there are two scales. In decimal (SI) units, each step is 1,000 times the previous one; “kilo” means thousand. In binary (IEC) units, each step is 1,024 times the previous one. The number 1,024 was not chosen at random; it is 2^{10} and very close to 1,000. Because memories work with binary addresses, their capacities are naturally powers of 2. For this reason “KB” was used for many years to mean 1,024 bytes as well, and this usage is still common today.

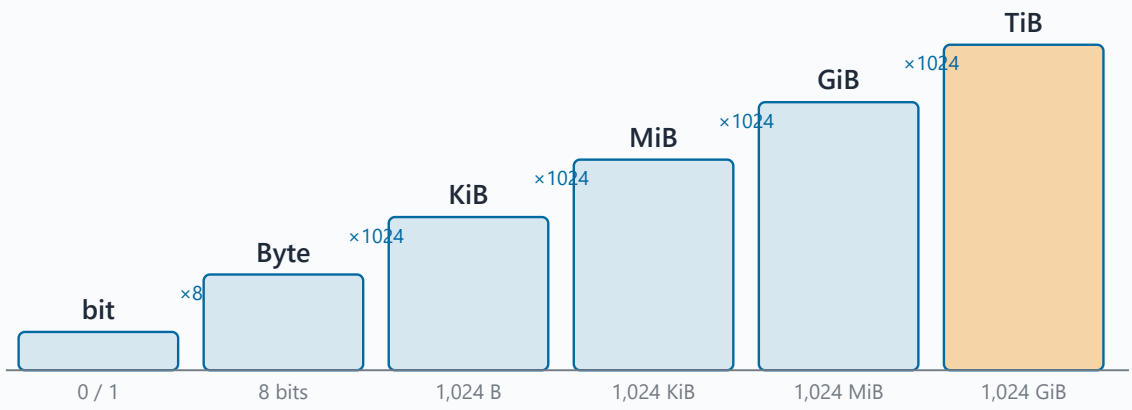


Figure 1.1.18: The ladder of binary (IEC) memory units. From bit to byte the factor is 8, and each following step grows by a factor of 1,024. The bar heights are not to scale; in reality each step is about a thousand times the previous one.

Familiar examples help in getting a feel for the size of the units. A single character is 1 byte. Half a page of plain text takes a few KB. A phone photo is about 3 MB, and an HD movie about 4 GB. Laptop disks are usually between 512 GB and 1 TB. The data centers of a large cloud service store petabytes of data.

Example 1.1.15

With binary units (1 KiB = 1,024 B, 1 MiB = 1,024 KiB, 1 GiB = 1,024 MiB):

1. **How many bits is 3 KiB?** *3 × 1,024 = 3,072* bytes; *3,072 × 8 = 24,576* bits.
2. **How many 4 MiB photos fit into a 128 GiB memory?** *128 × 1,024 = 131,072* MiB; *131,072 ÷ 4 = 32,768* photos. (The space taken by the operating system and applications is ignored in this calculation.)
3. **How many MiB is 5,000,000 bytes?** *5,000,000 ÷ 1,048,576 ≈ 4.77* MiB. In decimal units the same number is exactly 5 MB.

Going from a larger unit to a smaller one you multiply; going from a smaller one to a larger one you divide. The factor between byte and bit is 8; at every other step it is 1,024 for binary units and 1,000 for decimal units.

The small difference between 1,024 and 1,000 grows at larger units and causes confusion. To separate the two meanings, the International Electrotechnical Commission (IEC) gave the binary units the separate names in Table 1.1.11. 1,024 bytes is a **KiB** (kibibyte), 2^{20} bytes a **MiB** (mebibyte), and 2^{30} bytes a **GiB** (gibibyte). In this definition the prefixes KB, MB, GB keep their decimal meanings: 1 GB = 10^9 = 1,000,000,000 bytes. Disk manufacturers write capacity with this decimal (SI) definition. Operating systems often compute size in binary units but still display it as "GB."

Example 1.1.16

Why does a disk labeled "1 TB" on the box appear as about 931 GB in the operating system?

1. The manufacturer uses the decimal definition: 1 TB = 10^{12} = 1,000,000,000,000 bytes.
2. The operating system divides the byte count by 1 GiB = 2^{30} = 1,073,741,824 bytes: *1,000,000,000,000 ÷ 1,073,741,824 ≈ 931.3*.
3. The screen says "931 GB"; its correct name is "931 GiB."

Result: there are no missing bytes on the disk. The byte count is the same in both counts; only the definition of the unit differs. The difference grows as the unit gets larger: 2.4% at KB, 7.4% at GB, about 10% at TB.

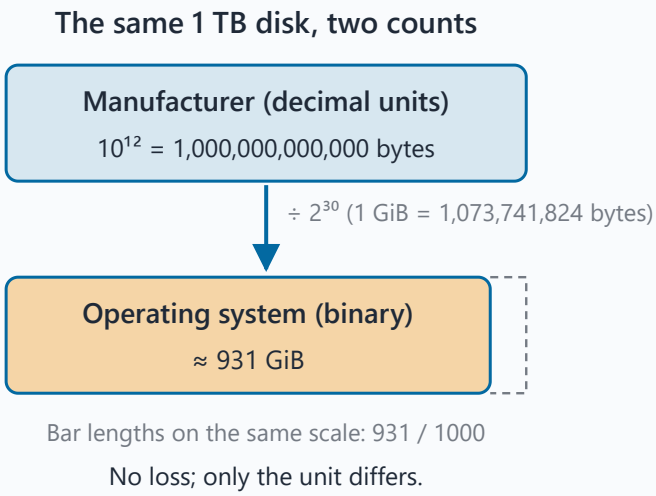


Figure 1.1.19: The same 1 TB disk, two counts. The manufacturer writes 10^{12} bytes; the operating system divides this by 2^{30} bytes and shows ≈ 931 GiB. The bar lengths are on the same scale: 931 / 1000.

Common mistake

Thinking the difference between 1 TB and 931 GB is "space lost in formatting" or "bad sectors." The space taken by formatting and the file system is very small. Almost all of the difference comes from the definition of the unit; the two numbers write the same amount of bytes in different units.

1.1.3.9 Bits or bytes? Speed and size

In unit abbreviations, uppercase and lowercase are not the same thing. A lowercase **b** stands for bit, an uppercase **B** for byte. Between 100 Mb (megabits) and 100 MB (megabytes) there is exactly a factor of 8. This distinction matters most when internet speed and file size are discussed together, because the two are traditionally measured in different units.

Definition 1.1.13

Data transfer rate is measured in *bits* per second: bit/s, kbit/s, Mbit/s (Mbps for short). The prefixes in speed units are decimal: 1 Mbps = 10^6 = 1,000,000 bit/s. **File and disk size**, on the other hand, is measured in *bytes*: KB, MB, GB.

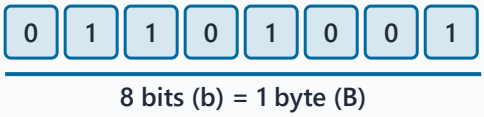


Figure 1.1.20: 8 bits (b) = 1 byte (B).

When calculating how long a file takes to download, the two units must be converted into the same language. Either the speed is converted to bytes (divided by 8) or the size is converted to bits (multiplied by 8). Both ways give the same result.

Example 1.1.17

With a connection of 100 Mbps, in how many seconds does a 3 GB file (3×10^9 bytes) download in the best case?

Step	Calculation	Result
Speed converted to bytes	100,000,000 ÷ 8	12,500,000 bytes/s = 12.5 MB/s
File converted to bits	3×10^9 bytes × 8	24×10^9 bits
Time	$24 \times 10^9 \div 10^8$	240 s = 4 minutes

The same result comes from the first row: *3,000,000,000 ÷ 12,500,000 = 240* s. If one thought "at 100 Mbps, 100 MB downloads per second," the time would come out as 30 seconds, which is an eighth of reality.

240 seconds is the best case. In reality the time is longer. Every piece of data sent over a network has addressing and control information added; this **protocol overhead** uses part of the line. Other devices sharing the same line divide the speed. The server itself can send data only at a limited rate. So on a 100 Mbps connection, download speeds of around 10–11 MB/s are usually seen.

Common mistake

Comparing the "11 MB/s" shown by the download program with the "100 Mbps" in the internet plan and concluding that the speed is low. 11 MB/s = 88 Mbps; given the overhead and sharing, this is a good value for a 100 Mbps line.

1.1.4 Data processing

A computer does not merely store data; it calculates with it and makes decisions. This section covers the basic kinds of operations the processor performs. It then shows, with the parity bit as an example, how the same operations are used to tell whether data has been corrupted in transmission.

1.1.4.1 Arithmetic and logical operations

Most of what programs do is a combination of four kinds of operations.

Kind	What it does	Operators	Type of result
Arithmetic	Calculates with numbers	+ − × ÷ %	Number
Comparison	Tests the relationship between two values	> < ≥ ≤ == !=	True / false
Logical	Combines conditions or reverses them	AND, OR, NOT	True / false
Decision	Chooses the path the program follows depending on the result	if … then	One path

Table 1.1.12: Basic kinds of operations.

Of the arithmetic operators, the % sign means not percent but the **remainder of a division**: *17 % 5 = 2*, because there are three 5s in 17 and 2 is left over. This is the same operation as the “mod” used in decimal-to-binary conversion. Among the comparison operators, == means “is equal?” and != means “is not equal?” Because a single = sign is used for assignment in programming languages, the question of equality is written with two equal signs.

The inputs and outputs of logical operations are true or false. In a computer, true is represented by 1 and false by 0. There are three basic logical operations.

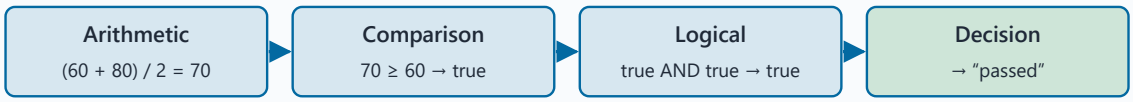
Definition 1.1.14

AND gives true if *both* conditions are true, and false if at least one is false. **OR** gives true if *at least one* of the conditions is true, and false if both are false. **NOT** reverses a single condition; it makes true false and false true.

A	B	A AND B	A OR B	NOT A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Table 1.1.13: Truth table of the AND, OR, and NOT operations (1 = true, 0 = false).

Table 1.1.13 covers every case in four rows, because two conditions can take only four combinations of values. In the AND column only the last row is 1: AND requires both sides to hold. In the OR column only the first row is 0: OR is satisfied if just one side holds. The everyday "and" and "or" are used in these senses too. "A student who passed the exam and has full attendance" requires both conditions. "A passenger who has an ID or a passport" finds either one sufficient.



Example: midterm 60, final 80, attendance_ok = true

Figure 1.1.21: The four kinds of operations in a grade calculation. For midterm 60, final 80, and attendance_ok = true: arithmetic (60 + 80) / 2 = 70; comparison 70 ≥ 60 → true; logical true AND true → true; decision: "passed."

Example 1.1.18

In a course the rule is as follows: average = (midterm + final) / 2; a student whose average is 60 or above *and* who meets the attendance requirement passes. Let us follow two students.

Step	Student 1	Student 2
Data	midterm 60, final 80, attendance_ok = true	midterm 40, final 90, attendance_ok = false
Arithmetic	(60 + 80) / 2 = 70	(40 + 90) / 2 = 65
Comparison	70 ≥ 60 → 1	65 ≥ 60 → 1
Logical	1 AND 1 → 1	1 AND 0 → 0
Decision	"passed"	"failed"

The second student's average is sufficient, but because the attendance condition is not met, the AND operation gives 0. If the rule had been written with "or" (1 OR 0 = 1), this student would have passed too. The choice of connective changes the meaning of the rule.

Figure 1.1.21 and Example 1.1.18 show how the four kinds of operations work as a chain. The arithmetic operation produces the number. The comparison turns this number into a true/false value. The logical operation combines several true/false values. The decision step then chooses the path according to the final value. The first three jobs are done in the ALU. In the decision step, the control unit selects the program's next instruction according to the result.

Common mistake

Confusing “greater than” with “greater than or equal to.” If a score of 60 is to count as passing, the condition must be “score ≥ 60,” not “score > 60.” If “score > 60” is written, a student who gets exactly 60 fails. Which side of the condition the boundary values fall on should always be checked separately.

1.1.4.2 Error detection with a parity bit

When data passes through a cable or a wireless link, electrical noise can corrupt some bits. A bit that is 1 can flip to 0, or a bit that is 0 to 1. How will the receiver know whether the bits that reached it are correct? The simplest solution is to add one extra bit to the data for checking purposes.

Definition 1.1.15

A **parity bit** is a check bit added to the data bits that makes the total number of 1s follow a predetermined rule. Under the **even parity** rule, the parity bit is chosen so that the total number of 1s in the data bits and the parity bit is even. If the number of 1s in the data bits is even, the parity bit is 0; if odd, it is 1. The receiver counts the 1s in all the bits; if the count is odd, there is an error.

A counterpart of this rule is **odd parity**: there the total number of 1s must be odd. The two rules are equally strong; it is enough that the sender and the receiver use the same one. The example below uses even parity.

Example 1.1.19

In ASCII the code of the letter “Y” is 89; in seven bits it is written 1011001. This letter is sent with even parity.

1. **The sender counts the 1s.** In 1011001 the 1s are in bits 1, 3, 4, and 7: four of them. Because four is even, the parity bit is 0. The eight bits sent: 10110010.
2. **One bit is corrupted in transmission.** Noise flips bit 3 from 1 to 0. The receiver gets 10010010.
3. **The receiver counts the 1s.** In 10010010 there are three 1s. Three is an odd number; the even-parity rule is violated. The receiver notices the error and asks for the data again.

Result: the single-bit error is caught. But the receiver knows only that an error *exists*; it cannot tell which bit was corrupted. So it cannot correct the error itself and asks for the data to be sent again.

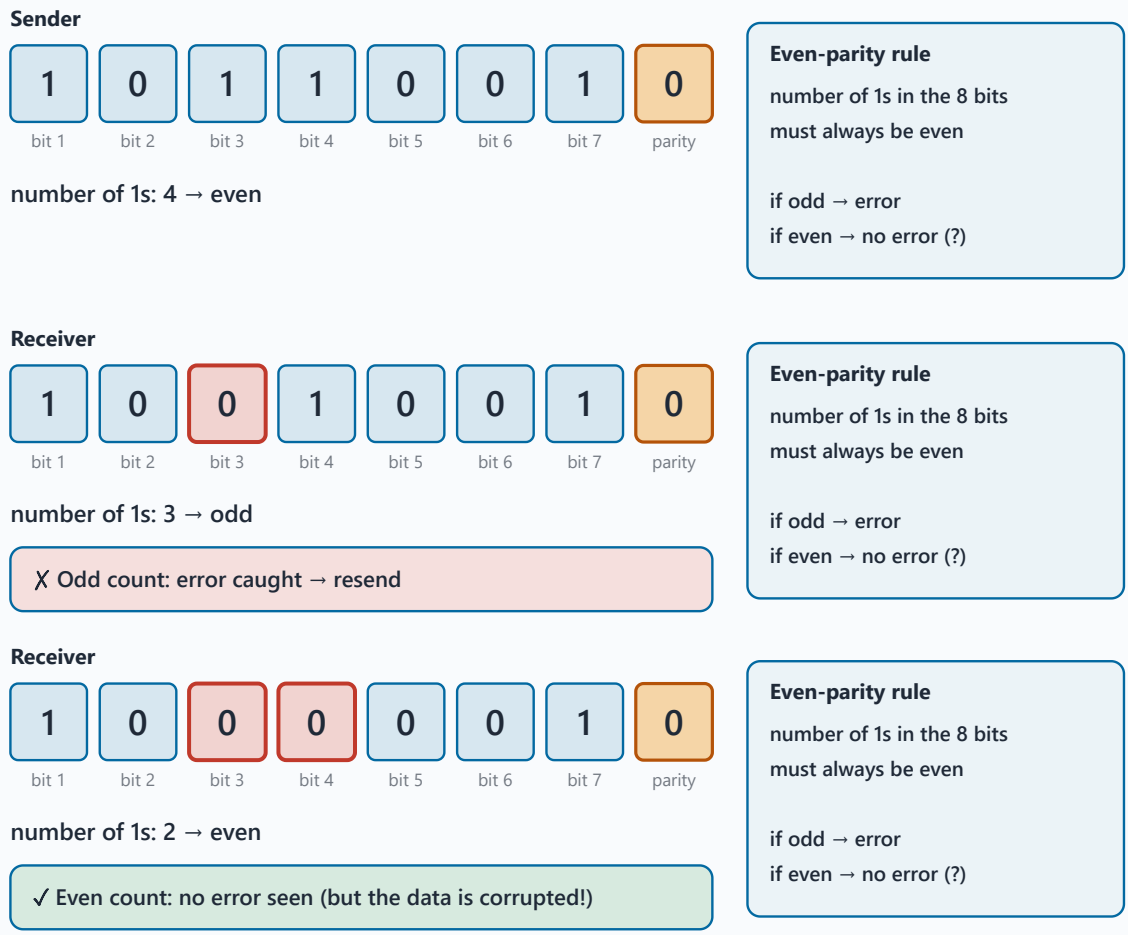


Figure 1.1.22: Error checking with even parity. Top, the sender: the seven data bits contain four 1s, and the parity bit is 0. Middle, the receiver: bit 3 is corrupted, the number of 1s is 3 (odd), and the error is caught. Bottom, two bits are corrupted at once: the number of 1s is 2 (even), the receiver sees no error, but the data is corrupted.

The last row of Figure 1.1.22 shows the limit of the parity bit. If bits 3 and 4 are corrupted together, the receiver gets 10000010. This sequence has two 1s; two is an even number. The receiver sees that the rule holds and takes the corrupted data for correct. The general rule is this: corrupting one bit raises or lowers the number of 1s by one, that is, it changes the odd/even state. Corrupting two bits changes the state twice and returns it to the start. For this reason a parity bit catches only an *odd number* of bit errors; an even number of errors goes unnoticed.

Corrupted bits	Received sequence	Number of 1s	Receiver's decision	Actual situation
None	10110010	4 (even)	No error	Correct
bit 3	10010010	3 (odd)	Error	Caught
Bits 3 and 4	10000010	2 (even)	No error	Missed
Bits 1, 3, and 4	00000010	1 (odd)	Error	Caught

Table 1.1.14: Result of the parity check by number of corrupted bits (sent 10110010, four 1s).

The parity bit is a very cheap check: a single extra bit for every seven or eight bits. It works in settings where errors are rare and come one at a time. In settings where noise corrupts bits in bursts, stronger methods are used. These methods add more check bits to catch multiple errors, and some can even locate the error and correct it. The error-correcting codes (ECC) used in memory modules and the checksums in internet packets are of this kind.

Common mistake

Using the *value* of the data bits (89) or the *number* of bits (7) when computing the parity bit. Parity looks only at the number of bits that are 1. 89, the value of 1011001, is odd, but the number of 1s in it is four, which is even; the parity bit is 0.

1.1.5 Kinds of computers

All computers use the same working cycle, but they are built in different forms for different jobs. A machine that runs a weather model and the control board of a washing machine are two very different computers working on the same principles. This section describes the main kinds of computers, the need each one meets, and where they differ from one another.

1.1.5.1 Classification by capacity and purpose

Computers are most often classified by two criteria: capacity (how much computing power, memory, and storage they offer) and purpose (whom they serve and for what job). Table 1.1.15 summarizes the main kinds by these two criteria.

Kind	Primary purpose	Typical example
Supercomputer	Intensive scientific computation	Climate model, physics simulation
Mainframe	A very large number of reliable transactions	Banking, insurance
Server	Service over a network	Website, e-mail, database
Personal computer	Individual use	Desktop, laptop
Embedded system	A specific function of a device	Air conditioner, car, washing machine
Thin client	Access to a central system	Library or bank branch terminal
Mobile device	Portable use	Phone, tablet
Quantum computer	Special classes of problems	Chemistry simulation, research

Table 1.1.15: Main kinds of computers and their typical uses.

This classification has no sharp boundaries. A powerful desktop computer acts as a server once a website is set up on it. A smartphone is both a mobile device and a personal computer. A car contains dozens of embedded computers, and they talk to one another over a network. Which kind a device falls into depends less on its hardware than on how it is being used at the moment.

1.1.5.2 Supercomputers and mainframes

At the top of the capacity scale there are two kinds. Both are very powerful and expensive, but each is designed to do a different thing best.

Definition 1.1.16

A **supercomputer** is a computer that solves a single large problem requiring very high computing power by splitting it across thousands of processors and working on the parts in parallel. A **mainframe** is a computer designed to run small transactions from a large number of users simultaneously, without interruption, and reliably.

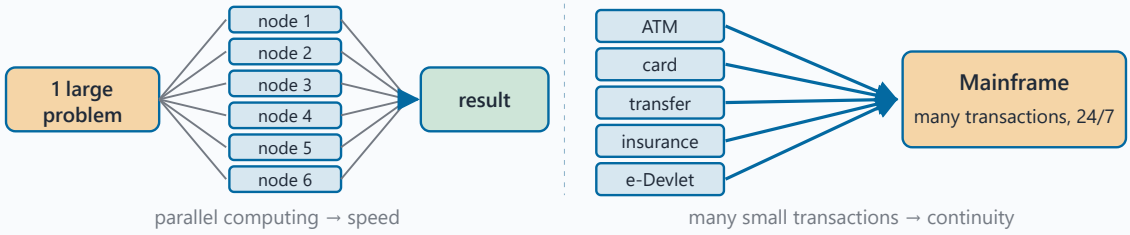


Figure 1.1.23: On the left, a supercomputer: one large problem is split across six nodes, the nodes compute in parallel, and the pieces combine into the result; the goal is speed. On the right, a mainframe: a large number of small jobs such as ATM, card, transfer, insurance, and e-Devlet transactions are processed 24/7; the goal is continuity.

Figure 1.1.23 shows the difference between the two kinds through their workloads. When a supercomputer is given a climate model, the Earth's atmosphere is divided into millions of small cells. Each processor computes the temperature, pressure, and wind of a few cells and shares its results with its neighbors. A job that would take months on a single computer finishes in a few hours. Supercomputers are also used in physics simulations, in drug and materials research, and in training large AI models. Their power is measured by the number of floating-point operations per second (FLOPS).

The mainframe's job is different. A bank receives thousands of card payments, transfers, and balance queries every second. Each of these transactions is tiny on its own; the hard part is to process all of them at once, without a single error, and without ever stopping. An hour's outage of a payment system causes large losses. Mainframes are therefore built with redundant hardware: when a part fails, the system keeps running without stopping, and the part can be replaced while it runs.

Priority	Supercomputer	Mainframe
Main goal	Computing speed	Transaction volume and continuity
Typical workload	Large scientific computation	A large number of enterprise transactions
Users	A small number of researchers	Thousands of simultaneous users

Priority	Supercomputer	Mainframe
Measure of success	Operations per second (FLOPS)	Transactions per second, uptime

Table 1.1.16: Comparison of a supercomputer and a mainframe.

Common mistake

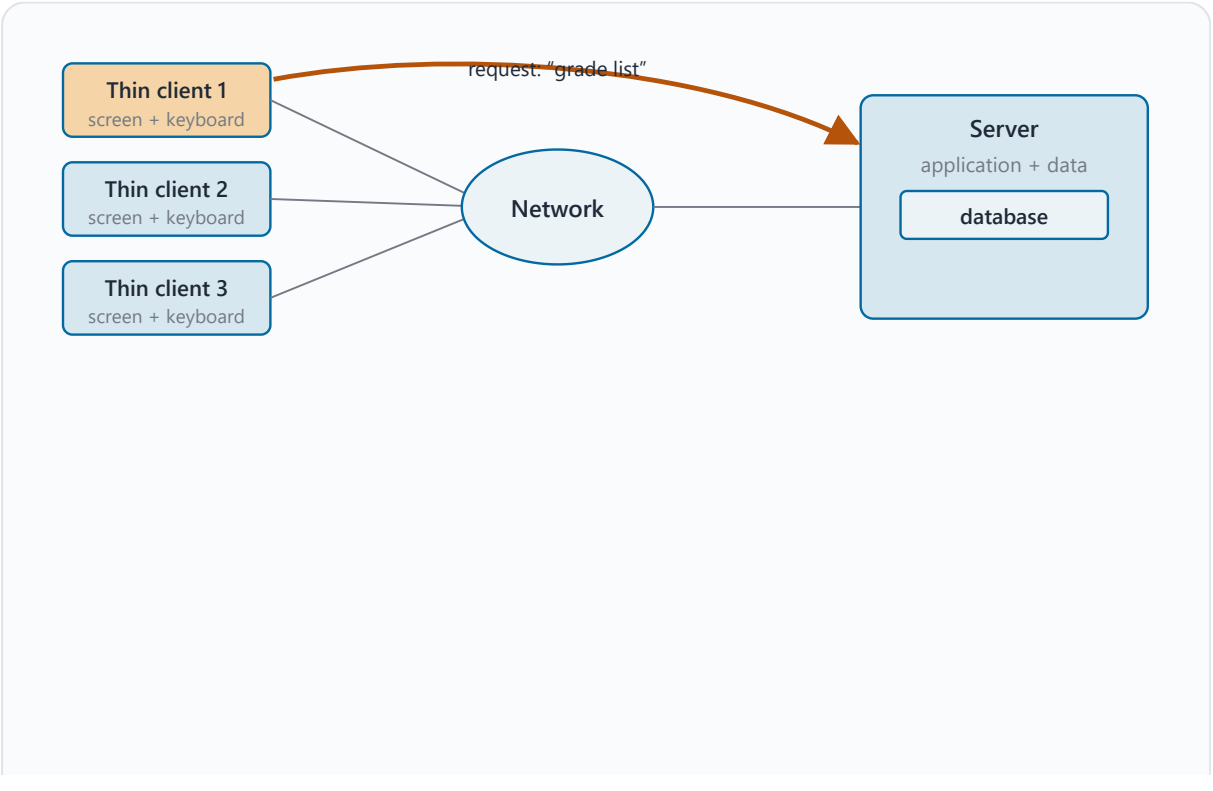
Taking a mainframe for an "old, slow supercomputer." The two kinds are not different points on the same scale; they are designed for different purposes. A supercomputer is not suitable for a bank's work, nor a mainframe for a climate model.

1.1.5.3 Servers and thin clients

Definition 1.1.17

A **server** is a computer that provides web, e-mail, file, database, or application services to other computers over a network. The computer that receives the service is called a **client**. A **thin client** is a simple client that leaves most of the processing to the server and does only display and input on its own hardware.

The conversation between a server and a client follows a **request–response** pattern. The client asks for something; the server processes the request and sends the result back. This pattern works both when a web page is opened and when an e-mail is downloaded.



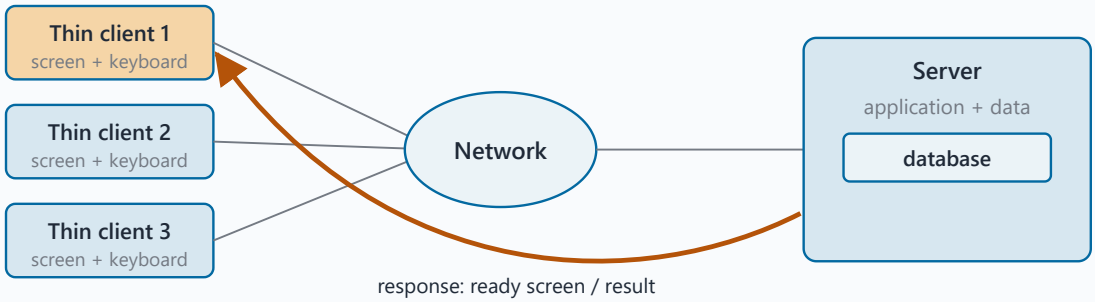


Figure 1.1.24: Three thin clients are connected to the same server over a network. Top, the request: client 1 asks for the grade list, and the request travels over the network to the server. Bottom, the response: the server queries the database and prepares the result, and the ready screen returns to the client.

Example 1.1.20

A grade list is opened from a thin client in a university library. Figure 1.1.24 shows three steps:

1. **Request.** Client 1 sends the "grade list" request to the server over the network. All the client does is turn the click on the screen into a request.
2. **Processing.** The real work is done on the server: the database is queried, the grades are found, the averages are computed, and the result is prepared as a screen.
3. **Response.** The ready screen returns to the client. The thin client only displays it and waits for the user's next click.

The working cycle is split between two computers here: input and output are on the client, processing and storage on the server.

The advantage of the thin client is ease of management. In a library with a hundred thin clients, programs are installed only on the server and updated only there. Because no data is kept on the clients, no data is lost when one breaks or is stolen; a new one is plugged in. The local hardware can be cheap and simple. The weakness is dependence: if the network or the server goes down, all hundred clients become useless at once.

Common mistake

Thinking of a server only as a big, special machine. "Server" is a role more than a type of hardware. A laptop at home is a server for the other computers while a file-sharing program runs on it. Servers in data centers, on the other hand, are built to take on this role continuously: powerful, redundant, and shaped to fit in racks.

1.1.5.4 Personal and mobile devices

A **personal computer** is a general-purpose computer designed for use by a single person. Personal and mobile devices are forms of the same general-purpose computer adapted to different conditions of use.

Device	Distinguishing feature	Trade-off
Desktop	Fixed use, expandable hardware, high performance	Not portable, uses more power
Laptop	Portable build, battery	Hardware upgrades are limited
Tablet	Touch screen, light use	Hard to write long texts or do heavy work
Phone	Mobile connectivity, sensors (GPS, accelerometer, camera), apps	Small screen, limited battery

Table 1.1.17: Comparison of personal and mobile devices.

On a desktop computer the case can be opened; the graphics card, memory, or disk can be replaced and upgraded. Thanks to plenty of power and ample cooling space, it gives the highest performance for the same price. A laptop fits the same parts into a small body and battery; part of the performance is sacrificed for portability and battery life. Tablets and phones go further in this direction: power consumption is kept so low that the processor runs without a cooling fan.

What really sets the phone apart from other personal devices is its sensors. The location receiver, accelerometer, gyroscope, camera, microphone, and fingerprint reader turn the phone into an input device that constantly measures the world. A map app showing direction, the screen rotating when the phone is turned sideways, and the step counter working are all thanks to these sensors.

Performance is not the only criterion in choosing a device. Portability, power consumption, screen size, and the main purpose of use are considered together. For a student who will write long reports and do programming, a laptop with a good keyboard and a large screen is more suitable than the fastest phone.

1.1.5.5 Embedded computers

Definition 1.1.18

An **embedded system** is a computer placed inside a larger device that carries out a specific task of that device. It is usually built around a **microcontroller**, which combines the processor, memory, and input–output units on a single chip.

Embedded computers are the most numerous kind of computer. A car has dozens of embedded computers that control the engine, the brakes, and the airbags. A washing machine, a microwave oven, an air conditioner, a smartwatch, a medical measuring device, and a traffic light each contain an

embedded computer. The common features of these computers are as follows:

- They do a single job, and their programs are loaded when the device is manufactured.
- They consume little power and work with limited memory and limited computing power.
- Reliability and **real-time** response matter: an airbag controller must detect a collision and decide within milliseconds.
- The user usually does not notice that there is a computer inside the device.

An embedded computer repeats its working cycle continuously as a **control loop**: measure, decide, act, measure again. In the air-conditioner example, the input unit is a temperature sensor, and the output unit is an **actuator** that switches the compressor on and off.

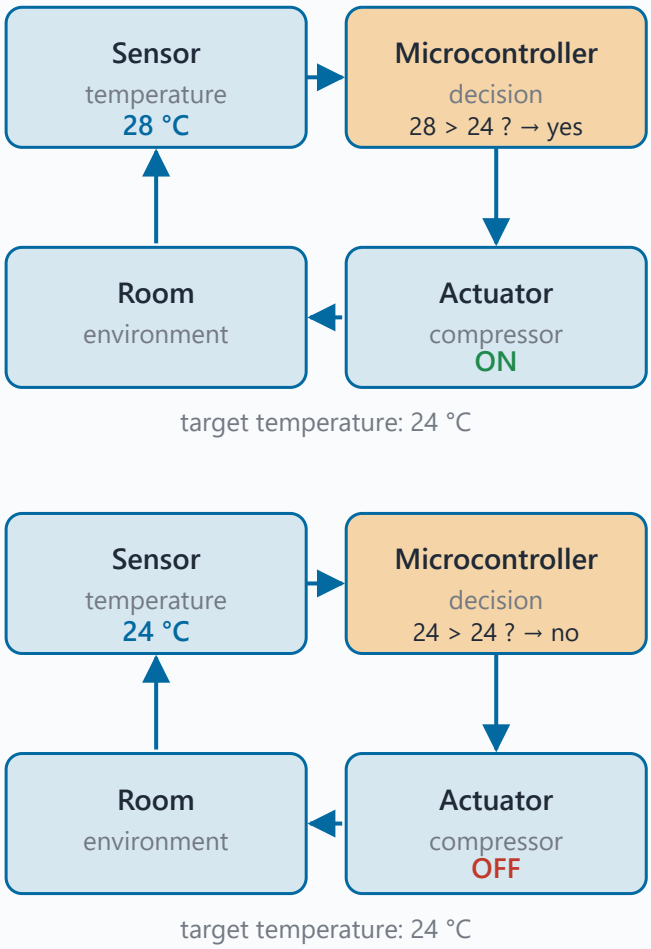


Figure 1.1.25: The control loop of an air conditioner; target temperature 24 °C. (a) The sensor measures 28 °C; because 28 > 24 is true, the microcontroller keeps the compressor on. (b) The sensor measures 24 °C; because 24 > 24 is false, the compressor switches off.

Example 1.1.21

The air conditioner's target temperature is 24 °C and the rule is: if the measured temperature is greater than the target, the compressor switches on; otherwise it switches off. The sensor measures 28, 26, and 24 °C in succession.

Round	Measurement (input)	Condition (processing)	Compressor (output)	Room
1	28 °C	28 > 24 → true	ON	cools
2	26 °C	26 > 24 → true	ON	cools
3	24 °C	24 > 24 → false	OFF	cooling stops

In the third round 24 is not greater than 24; it is equal to it. The "greater than" condition is false, and the compressor switches off. The loop does not end here. The sensor keeps measuring; when the room warms up and the temperature rises above 24 again, the compressor switches on again.

This simple rule has a flaw in practice. If the temperature drifts back and forth between 24.0 and 24.1, the compressor switches on and off too often and wears out. Real air conditioners therefore use two thresholds: for example, it switches on when the temperature passes 25 and off when it falls below 23. In the region in between, the compressor keeps its previous state. This small change shows that in embedded systems the software directly affects the life of the hardware.

Common mistake

Thinking that an embedded computer's loop runs once and ends. A calculator program gives its result and stops; an embedded control program keeps measuring and deciding without ever stopping for as long as the device is on.

1.1.5.6 Quantum computers

All the computers described so far process information with bits that are 0 or 1. Quantum computers use a different unit of information, based on the rules of quantum physics.

Definition 1.1.19

A **qubit** (quantum bit) is a unit of information that, before it is measured, can exist in a combination of the states 0 and 1. When a qubit is measured it gives 0 or 1; which one comes out is determined with a probability that depends on the weights in the combination. A **quantum computer** is a computer that calculates with qubits.

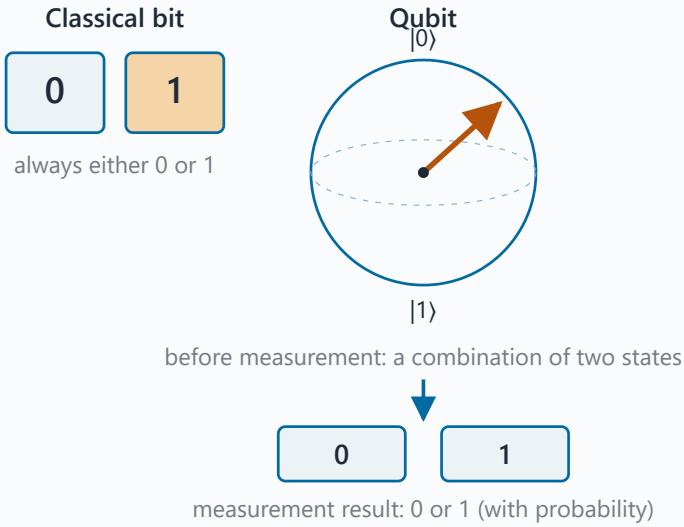


Figure 1.1.26: A classical bit is always either 0 or 1. Before measurement, a qubit is in a combination of the states |0⟩ and |1⟩; when measured, it gives 0 or 1 with some probability.

For certain classes of problems, quantum computers offer computing methods very different from classical computers. Simulating the behavior of molecules and materials, certain optimization problems, and breaking some encryption methods are studied among these classes. The problem of factoring large numbers into primes in particular is watched closely, because it is the problem on whose difficulty the security of some widely used encryption systems rests today.

That said, quantum computers have not replaced classical computers in everyday use, and they are not expected to replace them in general-purpose use in the near future. Qubits are disturbed by the slightest influence from the environment; for this reason most quantum computers are operated cooled to temperatures close to absolute zero. For tasks such as word processing, browsing the internet, or calculating a spreadsheet, a quantum computer has no advantage at all. A quantum computer is not a faster version of a classical computer but a different tool designed for particular problems.

Common mistake

Thinking that a qubit “tries all the answers at the same time and picks the right one.” A measurement always gives a single result. Quantum algorithms are carefully designed to lower the probability of wrong answers and raise the probability of the right one; this is possible only for certain problems.

1.1.6 Information technology

The previous sections looked inside the computer: at bits, operations, and kinds of hardware. This section looks outward, at the systems that computers form together with people and organizations. The whole of these systems, which turn data into information, is called information technology.

1.1.6.1 What is information technology?

Definition 1.1.20

Information technology (IT) is the whole of the systems used to collect, process, store, and transmit data. An IT system has four technical components: **hardware** (physical devices), **software** (programs and operating systems), **data** (the raw values being processed), and **communication** (networks and data transfer).

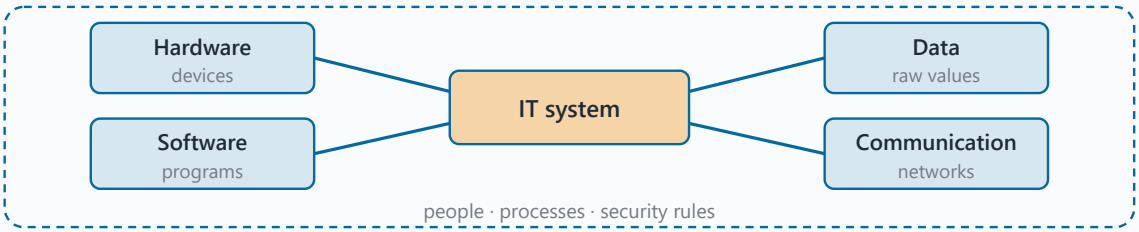


Figure 1.1.27: An IT system. In the middle are the four technical components (hardware, software, data, communication); around all of them are people, processes, and security rules.

Figure 1.1.27 stresses that an IT system does not consist of computers alone. Consider the patient record system of a hospital. The hardware is the servers, computers, and printers. The software is the record program and the database. The data is patient information, test results, and prescriptions. Communication is the network between buildings. But the system does not work without the secretaries who enter the records, the doctors who read them, and the rules that decide who may see which information. The people, processes, and security rules in the outer ring of the system are as important as the technical components. Even the best hardware is not secure if a user writes the password on paper and sticks it to the screen.

1.1.6.2 The difference between data and information

In everyday language, data and information are mostly used with the same meaning. In computer science there is a clear difference between them, and this difference explains the reason for the existence of every IT system.

Definition 1.1.21

Data is a raw measurement, number, text, or record; without context its meaning may remain unclear. **Information** is a meaningful result, useful for making decisions, that emerges when data is organized, processed, and interpreted in a context.

Example 1.1.22

A piece of paper has only these three numbers on it: 38, 41, 39. How do these numbers become information?

1. **Raw data.** 38, 41, 39 are only numbers. They have no meaning unless it is known what was measured: they could be ages, exam scores, or speeds.
2. **Context.** It is added that they are three temperature readings of a patient during the day, and the unit is °C. Now what the numbers show is clear.
3. **Processing.** The average is computed: *(38 + 41 + 39) / 3 = 118 / 3 ≈ 39.3*.
4. **Information.** "The patient's average temperature is 39.3 °C; the fever is high." A decision can be based on this sentence: the patient is given a fever reducer.

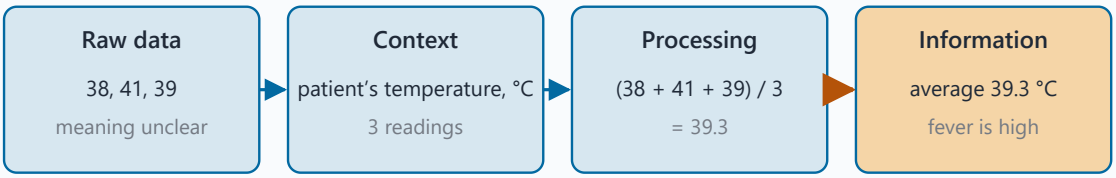


Figure 1.1.28: Four steps from data to information: raw data (38, 41, 39; meaning unclear), context (patient's temperature, °C, 3 readings), processing (average 39.3), and information (average 39.3 °C, fever is high).

Figure 1.1.28 gathers the steps of the conversion on a single line. The computer does only the processing part of these steps: it adds and divides. It is the human who knows that the numbers are temperature readings and that 39.3 °C should count as "high." Without the human who adds the context and interprets the result, the computer's output is still only a number. If the same 39.3 had been a room temperature, the interpretation would be entirely different.

Common mistake

Thinking that the more data is collected, the more information is obtained. Data whose context is unclear, disorganized, or erroneous gives no information however much of it there is; it can even lead to wrong interpretations. Information comes not from the amount of data but from organizing and interpreting it correctly.

1.1.6.3 Information technology in everyday life

Information technology is part of everyday tasks today in almost every field. A few examples show how the same working cycle looks in different fields.

- **Education:** online courses, digital course materials, and learning platforms. A student information system collects grades (input), computes averages (processing), shows the transcript (output), and keeps the records for years (storage).
- **Health:** electronic patient records, imaging systems (X-ray, MRI), and patient monitoring. A patient's test results can be accessed in the same record from different hospitals.
- **Finance:** banking, payment systems, and transaction security. A card payment is approved within seconds by passing through computers in several countries.
- **Business and entertainment:** corporate communication, automation of business processes, games, and media services. A music app recommends new songs based on the songs a listener has played.

IT contributes in a similar way in all of these fields: collecting data faster, processing it more accurately, keeping it longer, and delivering it sooner to the person who needs it.

1.1.6.4 The data life cycle

Data does not wander at random in an IT system; it passes through certain stages. The whole of these stages is called the **data life cycle**.

Definition 1.1.22

The data life cycle has four stages. **Collection**: data is taken from people, machines, or sensors. **Processing**: erroneous data is filtered out and calculations are made. **Storage**: data is kept in a file, a database, or the cloud. **Presentation**: the result is shared as a report, table, chart, or application screen. Decisions made on the basis of the presented result start the collection of new data, and the cycle turns again.

Example 1.1.23

The sensor of a weather station sends four temperature readings: 21.4 · −99 · 22.1 · 23 (°C). When the sensor cannot take a measurement, it sends −99 as a "fault" code.

1. **Collection.** The four values are received from the sensor.
2. **Processing.** −99 is not a real temperature but a fault code; it is discarded. The average of the remaining three readings is taken: *(21.4 + 22.1 + 23) / 3 = 66.5 / 3 ≈ 22.17*, that is, about 22.2 °C.
3. **Storage.** The readings and the average are saved to the database together with the time.

4. **Presentation.** A daily temperature chart is published on a web page. A farmer who follows the chart sets the irrigation time accordingly; this decision is tested as new readings are collected.

If −99 had not been discarded, the average would have been *(21.4 − 99 + 22.1 + 23) / 4 = −32.5 / 4 ≈ −8.1* °C. This number means nothing in midsummer. Filtering at the processing stage is essential for data to become information.

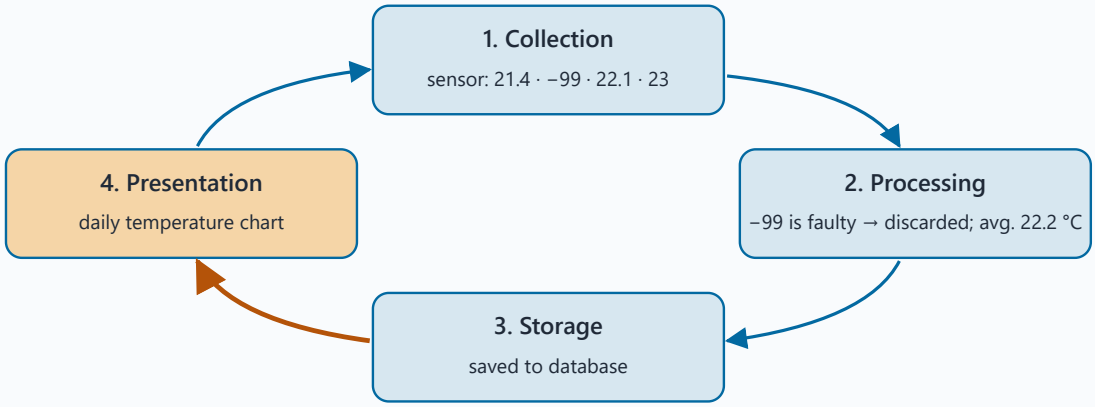


Figure 1.1.29: The data life cycle in the weather-station example: collection (sensor: 21.4 · −99 · 22.1 · 23), processing (−99 is discarded; average 22.2 °C), storage (saved to the database), and presentation (daily temperature chart).

Figure 1.1.29 shows that the cycle is a closed ring. The data life cycle closely resembles the computer's input–processing–output–storage cycle, but its scale is different. The computer's cycle turns on a single machine within a thousandth of a second. The data life cycle spreads across all of an organization's systems and can take days, months, even years.

Common mistake

Writing zero instead of "ignoring" the faulty data. If −99 is counted as 0 instead of being discarded, the average comes out as *(21.4 + 0 + 22.1 + 23) / 4 ≈ 16.6* °C, which is still wrong. A missing or faulty reading must not enter the average at all; the divisor is also the number of valid readings, not the number of readings.

1.1.6.5 Benefits and responsibilities

Information technology brings important conveniences:

- **Speed and automation:** data is processed millions of times faster than it could be by hand; repetitive tasks, such as computing the averages of all students, are done automatically.

- **Access and communication:** it becomes easier to reach and share information from different places; a doctor can see the test results of a patient in another city.

The same conveniences also create new responsibilities:

- **Privacy and security:** personal data must be protected against unauthorized access and loss. A leak of a health record or bank details harms the person for a long time. In Türkiye, the processing of personal data is regulated by Law No. 6698 on the Protection of Personal Data (KVKK).
- **Accuracy and ethics:** wrong data and biased systems can produce faulty results. Software that learns from biased decisions in the past can repeat the same bias quickly and at large scale.

Security is therefore not only a technical matter. Even the strongest encryption gives no protection if a user types their password into a fake e-mail. The security of an IT system depends on hardware and software, but also on the knowledge of the people who use it, on the organization's rules, and on how well those rules are applied. Technological convenience does not remove the obligation to use data correctly, securely, and responsibly.

1.1.6.6 The future of information technology

Today, four areas visibly drive the development of information technology.

Area	Short definition
Artificial intelligence	Decision-support and generative systems that learn patterns from data
Internet of Things	Devices with sensors sharing data over a network
Cloud computing	Providing computing and storage resources over a network
Big data analytics	Drawing meaningful results from high-volume and varied data

Table 1.1.18: Four prominent areas in information technology.

These four areas often work together in a single system. Figure 1.1.30 illustrates this with a smart irrigation system.

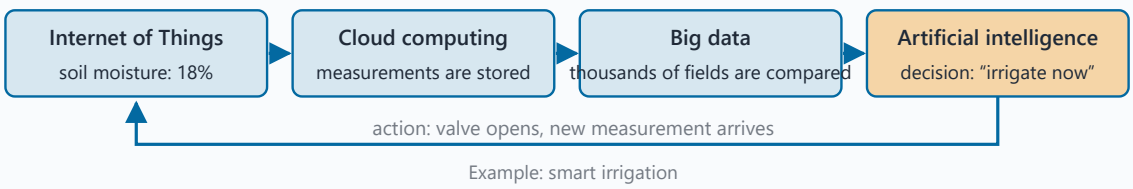


Figure 1.1.30: Smart irrigation: the Internet of Things sensor measures soil moisture (18%), the cloud stores the measurements, big data analytics compares thousands of fields, artificial intelligence makes the "irrigate now" decision; the valve opens and a new measurement arrives.

The sensor in the field is part of the Internet of Things: it measures soil moisture and sends it over the network. The measurements are stored in the cloud, that is, not on the farmer's own computer but on servers rented over the network. Big data analytics compares the moisture, weather, and crop data of thousands of fields in the same region. Artificial intelligence, based on what it has learned from this comparison, makes the "irrigate now" decision, and the valve opens. After irrigation, new measurements arrive and the cycle starts again. In essence this system, too, is a large-scale application of the data life cycle and of the computer's basic working cycle.

New technologies bring new questions with them. The energy consumed by data centers keeps rising. People without access to technology can be left behind as services go digital. Connecting more devices to the network widens the attack surface. Who is to answer for the decisions made by artificial intelligence is still debated. The future of information technology will be shaped not only by technical progress but also by the answers given to questions of energy, accessibility, security, and ethics.

1.1.7 Summary

- A **computer** is an electronic system that processes the data it receives according to program instructions. In every task it uses the same four stages: **input**, **processing**, **output**, and **storage**. Storage talks to processing in both directions: results are written and read back when needed.
- Input units turn physical events into bits. In the processor, the **control unit** manages the order of instructions, the **ALU** does calculations and comparisons, and the **registers** hold the values being worked on.
- **RAM** is fast but volatile; its contents are erased when the power goes off. **SSDs** and hard disks are non-volatile. "Save" copies the document from RAM to non-volatile storage.
- In a computer, all kinds of data are represented by sequences of **bits**; 8 bits make 1 **byte**. The meaning of a bit sequence is determined by the rule (the encoding) used to read it.
- n bits represent 2^n different values (unsigned $0 \ldots 2^n - 1$). For N values you need the smallest n with $2^n \geq N$: 6 bits for 58 letters.
- To convert binary to decimal, add the **place values** of the bits that are 1. To convert decimal to binary, divide the number by 2 repeatedly and read the remainders from last to first: $45 = 101101_2$. Each **hexadecimal** digit corresponds to four bits: $45 = 2D_{16}$.
- A transistor is a two-state switch; conducting is read as 1, cutoff as 0. Because telling two states apart is reliable, computers work in binary.
- Text is stored with **ASCII** or **Unicode** codes, and images with pixels that have **RGB** components. The size of an uncompressed image is width × height × bit depth ÷ 8 bytes.
- Sound becomes numbers through **sampling**, **quantization**, and **encoding**; size = sampling rate × bit depth × channels × duration. One minute of CD quality is ≈ 10.6 MB (10.1 MiB). Video consists of frames shown at a certain number per second (fps).

- Compression can be **lossless** (ZIP, PNG; the original data comes back exactly) or **lossy** (JPEG, MP3; detail is discarded). **RLE** writes consecutive repetitions as (length, value) pairs; it shrinks data with long repetitions and enlarges data that changes often.
- Beyond the byte, decimal (SI) units grow by factors of 1,000 (KB, MB, GB) and binary (IEC) units by factors of 1,024 (KiB, MiB, GiB). Disk makers use SI, while operating systems often show the binary value labeled "GB"; this is why 1 TB appears as ≈ 931 GiB. Network speed is measured in **bits per second** and file size in **bytes**; 100 Mbps = 12.5 MB/s.
- **AND** requires both conditions, **OR** at least one; **NOT** reverses a condition. Arithmetic, comparison, logical operations, and the decision work as a chain.
- A **parity bit** catches an odd number of bit errors but cannot locate them; an even number of errors goes unnoticed.
- Kinds of computers are told apart by purpose, capacity, and environment: a supercomputer is designed for speed, a mainframe for continuity, a server for network service, and an embedded system for the task of a single device. A quantum computer is a different tool for particular problems.
- **Information technology** consists of hardware, software, data, and communication together with people, processes, and security rules. The computer processes data; it is the human who adds context and turns it into **information**. Data passes through a **life cycle** of collection, processing, storage, and presentation.

Question	Rule	Example
How many values do n bits give?	2^n	8 bits → 256
How many bits for N values?	Smallest n with $2^n \geq N$	58 → 6 bits
Binary to decimal	Sum of the place values of the bits that are 1	$01000001_2 = 64 + 1 = 65$
Decimal to binary	Repeated division by 2; remainders read from last to first	$45 = 101101_2$
Binary to hexadecimal	Group in fours from the right	$0010\ 1101_2 = 2D_{16}$
Image size	width × height × bit depth ÷ 8	800 × 600, 24 bits → 1,440,000 bytes
Sound size	sampling rate × bit depth × channels × duration ÷ 8	CD, 1 min → 10,584,000 bytes
Download time	size (bits) ÷ speed (bit/s)	3 GB, 100 Mbps → 240 s

Table 1.1.19: Calculation rules used in these notes.

1.1.8 Key terms

computer data information input unit output unit storage central processing unit (CPU) control unit arithmetic logic unit (ALU) register random access memory (RAM) volatile / non-volatile bit byte binary decimal hexadecimal digit / place value transistor

character set ASCII Unicode encoding pixel resolution color depth sampling / sample sampling rate quantization frame lossless compression lossy compression run-length encoding (RLE) kilobyte / megabyte / gigabyte / terabyte kibibyte / mebibyte / gibibyte arithmetic operation comparison logical operation parity bit supercomputer mainframe server client thin client request / response personal computer mobile device embedded system microcontroller sensor quantum computer qubit (quantum bit) information technology (IT) data life cycle privacy artificial intelligence Internet of Things cloud computing big data

1.1.9 Test Yourself

1. What is 10110110_2 in decimal? What is its hexadecimal form?

   Answer

   The place values of the bits that are 1 are 128, 32, 16, 4, and 2: *128 + 32 + 16 + 4 + 2 = 182*. The groups of four are 1011 = B and 0110 = 6; the hexadecimal form is $B6_{16}$. Check: *11 × 16 + 6 = 182*.

2. Which sequence results when 100 is converted to binary by repeated division by 2? How is the result checked with place values?

   Answer

   100 ÷ 2 = 50 remainder 0; 50 ÷ 2 = 25 remainder 0; 25 ÷ 2 = 12 remainder 1; 12 ÷ 2 = 6 remainder 0; 6 ÷ 2 = 3 remainder 0; 3 ÷ 2 = 1 remainder 1; 1 ÷ 2 = 0 remainder 1. The remainders read from last to first give 1100100_2. Check: *64 + 32 + 4 = 100*.

3. A faculty has 100 laboratory lockers numbered 0 to 99. What is the minimum number of bits needed to encode each locker's number in binary? How many codes are left unused?

   Answer

   The number of values to be coded is 100. Since $2^6 = 64 < 100$, 6 bits are not enough; since $2^7 = 128 \geq 100$, 7 bits are needed. $128 - 100 = 28$ codes are left unused.

4. How many bytes does an uncompressed 640 × 480 image with 24-bit color take? How many KB

and KiB is that? What is the size if the same image is converted to 8-bit grayscale?

Answer

The number of pixels is *640 × 480 = 307,200*. Since 24 bits = 3 bytes, *307,200 × 3 = 921,600* bytes ≈ 922 KB; *921,600 ÷ 1,024 = 900* KiB. At 8 bits, 1 byte per pixel remains: 307,200 bytes = 300 KiB (≈ 307 KB), that is, one third.

5. How many bytes does a 30-second *mono* (single-channel) audio recording take, made with a 44,100 Hz sampling rate and 16-bit depth?

Answer

44,100 × 16 × 1 × 30 = 21,168,000 bits; ÷ 8 = 2,646,000 bytes ≈ 2.65 MB (≈ 2.52 MiB if 1 MiB = 1,048,576 bytes is used). If the recording were stereo, the size would double to 5,292,000 bytes.

6. How is the row WWWBBBBBBBWW (W = white, B = blue; each pixel 1 byte) encoded with RLE? How many bytes are saved?

Answer

The runs are three white, seven blue, and two white: (3, W) (7, B) (2, W). Three pairs × 2 bytes = 6 bytes; the original row is 12 bytes. 6 bytes are saved and the row is halved. The sum of the lengths is *3 + 7 + 2 = 12*.

7. Seven data bits 1101000 are sent with even parity. What is the parity bit? If 11000001 reaches the receiver, what does the receiver decide?

Answer

1101000 has three 1s; to make the total even the parity bit is 1, and the sequence sent is 11010001. The received 11000001 has three 1s; the count is odd, so the receiver knows there is an error and asks for the data again. (In reality bit 4 flipped from 1 to 0; the receiver cannot tell where.)

8. With a 50 Mbps connection, in how many seconds does a 1.5 GB (1.5×10^9 bytes) file download in the best case?

Answer

The file is converted to bits: $1.5 \times 10^9 \times 8 = 12 \times 10^9$ bits. The speed is 50×10^6 bit/s. The time is $12 \times 10^9 \div (50 \times 10^6) = 240$ s = 4 minutes. It can also be found by converting the speed to bytes: 50 Mbps = 6.25 MB/s; 1,500 MB ÷ 6.25 = 240 s.

9. About how many GiB does a disk labeled “500 GB” on the box show in the operating system?

   Answer

   The manufacturer uses decimal units: 500×10^9 bytes. The operating system divides this by 2^{30} = 1,073,741,824 bytes: ≈ 465.7 GiB. No bytes are lost; the difference comes from the definition of the unit.

10. A greenhouse’s ventilation turns on with the condition (temperature > 28) AND (humidity < 60). When the temperature is 30 °C and the humidity is 70%, what is the value of the condition? What would happen if OR were written instead of AND?

    Answer

    30 > 28 is true (1); 70 < 60 is false (0). 1 AND 0 = 0: the ventilation does not turn on. With OR, 1 OR 0 = 1 and the ventilation turns on. With the same data, the choice of connective changes the decision.

1.1.10 Further reading

- J. G. Brookshear and D. Brylow, *Computer Science: An Overview* (13th ed., Pearson, 2019), Chapter 1: the storage of bits, the binary system, the representation of text, images, and sound, data compression, and transmission errors are covered in an order very close to these notes and with plenty of examples.
- W. Stallings, *Computer Organization and Architecture* (11th ed., Pearson, 2019), Chapters 1–2: the basic structure of a computer, the parts of the processor, and the historical development of computers are covered from a more detailed hardware viewpoint.
- C. Petzold, *Code: The Hidden Language of Computer Hardware and Software* (2nd ed., Microsoft Press, 2022): starting from Morse code and light bulbs, it builds up binary numbers, logic gates, and the processor without requiring technical background.

1.1.11 References

1. J. G. Brookshear, D. Brylow, *Computer Science: An Overview*, Pearson.
2. W. Stallings, *Computer Organization and Architecture*, Pearson.
3. The Unicode Consortium, *The Unicode Standard*.
4. NIST, *Glossary of Key Information Security Terms* (NISTIR 7298).
5. IEC 80000-13, *Quantities and units — Part 13: Information science and technology* (binary prefixes: kibi, mebi, gibi).
6. Law No. 6698 on the Protection of Personal Data, Official Gazette, 7 April 2016.

PART I · CHAPTER 1.2

Computer Hardware

Every program written on a computer ultimately runs on physical parts you can touch. The processor executes instructions, memory holds the data in use at that moment, and the storage device keeps data permanently; the keyboard and the screen form the bridge between the user and the machine. These notes explain what each of these parts does, how they communicate with one another, and which numbers to look at when choosing a computer. Most topics are supported by small calculations: the length of a clock cycle from the clock speed, the time a cache saves, the pixel density of a screen, how many seconds a file takes to copy, and how many watts a power supply must deliver. These calculations make it easier to see what the numbers in technical specifications mean.

LEARNING OUTCOMES

Components
Explain the functions of the basic hardware parts (processor, motherboard, RAM, ROM, graphics card, power supply, case).

Comparison
Distinguish RAM, ROM, HDD, and SSD within the memory hierarchy; show, by counting hits and misses, why a cache speeds things up.

Choosing devices
Choose input and output devices according to their purpose; calculate and interpret values such as resolution, pixel density, refresh rate, and storage speed.

Compatibility
Evaluate whether components fit together in terms of connection, power, physical size, and heat.

1.2.1 What is hardware?

When people hear “computer,” most first think of the screen, the keyboard, and the case. These are the visible face of the computer. Inside the case are the processor, the memory modules, the storage device, the power supply, and the motherboard that connects them all. This section draws the general

picture of these parts. Each part is treated separately in the following sections.

1.2.1.1 Hardware and software

A computer system consists of two layers. The first layer is the physical parts: things you can touch, plug in and unplug, and replace when they break. The second layer is the instructions that tell these parts what to do. The first layer is called hardware, the second software.

Definition 1.2.1

Hardware is all the physical parts of a computer system: the processor, memory, storage devices, motherboard, power supply, case, input and output devices, and the cables that connect them. **Software** is the programs that determine which operations these parts perform and in what order.

Hardware and software complete each other. Without software, hardware does nothing; even the fastest processor waits when it is given no instructions. Without hardware, software cannot run either, because a physical medium is needed to execute the instructions, hold the data, and show the result. It is like the relationship between a music player and a song: the player is hardware, and the notes of the song are software.

Every task a computer performs consists of five basic steps. Figure 1.2.1 shows these steps and the hardware part that handles each of them:

- **Input devices** bring data into the system. A key pressed on the keyboard, a movement of the mouse, and sound reaching the microphone are each an input.
- The **CPU** (central processing unit) processes instructions: it adds, compares, and decides.
- **RAM** holds the running program and the data in use at that moment. The processor reads data from here and writes results back here.
- The **storage** device (SSD or HDD) keeps data permanently. Its contents are preserved even when the computer is switched off.
- **Output devices** present the result to the user: an image on the screen, a sound from the speaker, a page from the printer.

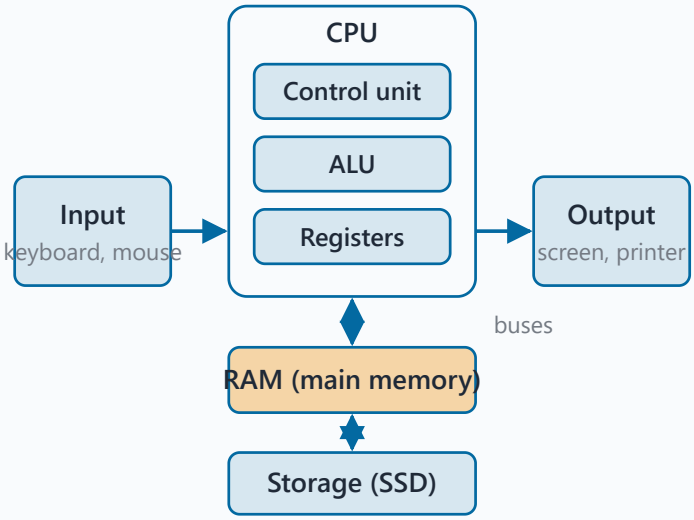


Figure 1.2.1: The flow of data in a computer. Data from the input devices reaches the CPU. Inside the CPU are the control unit, the arithmetic logic unit (ALU), and the registers. The CPU constantly exchanges data with RAM; RAM works in both directions with the storage device. The result is sent to the output devices. The connections between the parts are called buses.

The two-way arrows in Figure 1.2.1 are important. Traffic between the CPU and RAM runs both ways: the processor reads both instructions and data from RAM and writes its results to RAM. Between RAM and storage, too, data flows both ways. When a program is opened, its file is copied from storage to RAM; when a document is saved, the changes are written from RAM to storage. This flow is followed step by step in 1.2.9.1 on a document being opened and saved.

Example 1.2.1

In a calculator program, `12 + 30` is typed and Enter is pressed. The five steps run as follows:

1. **Input.** The keyboard turns each key press into a numeric code and sends it to the computer: first `1`, `2`, then `+`, `3`, `0`, and Enter.
2. **Memory.** The calculator program is already in RAM. The pressed keys are also written to the program's variables in RAM.
3. **Processing.** When Enter arrives, the CPU executes the program's addition instruction. It moves the numbers 12 and 30 from RAM into registers, adds them in the ALU, and gets the result 42.
4. **Output.** The program sends 42 to the graphics card to be displayed; the monitor shows the result.
5. **Storage.** No permanent record is needed in this example. If the program keeps past calculations in a file, the result is also written to the SSD.

As can be seen, even the simplest task requires several hardware parts to work together.

Common mistake

Taking “memory” and “storage” to be the same thing. In everyday speech both may be called “memory,” but their jobs differ. RAM is the temporary space the program uses *while running*, and it is emptied when the power goes off. Storage (SSD, HDD) is where files stay *permanently*. In the sentence “the computer has 16 GB of memory and 512 GB of storage,” two separate parts are meant.

1.2.1.2 Hardware groups

Computer hardware can be arranged into six groups by function. This grouping helps one quickly see what role a part plays in the system.

Group	Examples	Function
System components	CPU, motherboard, RAM, GPU, power supply, case	processes instructions, connects and powers the parts
Input devices	keyboard, mouse, microphone, camera, sensors	bring data into the system
Output devices	monitor, printer, speaker, projector	present the result to the user
Storage	HDD, SSD, USB flash drive, memory card, optical disc	keeps data permanently
Network hardware	network card, switch, router, access point	connects computers to one another
Connections	USB, HDMI, DisplayPort, audio and network ports	physically connect devices to the system

Table 1.2.1: Hardware groups and examples.

A few names in network hardware should be explained briefly. A **network interface card** (NIC) lets the computer connect to a wired or wireless network. A **switch** is the box that connects the devices on the same local network. A **router** directs traffic between different networks, for example between a home network and the internet. An **access point** lets wireless devices join the wired network.

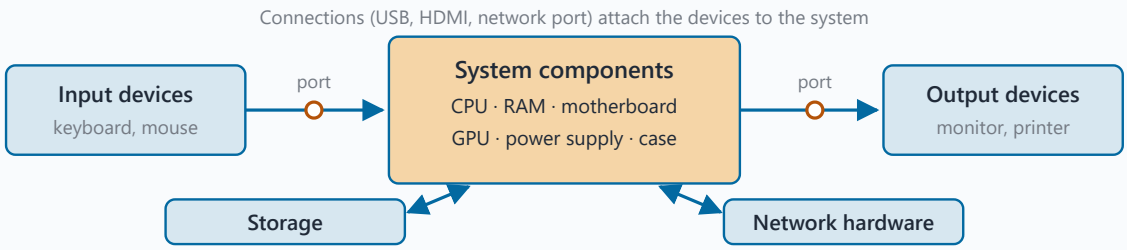


Figure 1.2.2: How the hardware groups relate. Input devices give data to the system components; the system components send the result to the output devices. Storage and network hardware work in both directions with the system. Devices are attached to the system through ports.

Figure 1.2.2 summarizes the flow of data between the groups. Input and output devices are one-way: one brings data in, the other sends it out. Storage and network hardware both read and write. A file is read from an SSD, and a file is written to the same SSD. A network card both receives and sends data. This distinction is the basis of the input–output classification in 1.2.8.1.

1.2.2 The central processing unit

The processor is the part of the computer that executes instructions. Every line of a program ends up as very simple instructions the processor understands: add two numbers, read a value from memory, compare two values, jump to another instruction depending on the result. The processor's power comes from being able to execute these simple instructions billions of times per second.

1.2.2.1 The processor's job: fetch, decode, execute

Definition 1.2.2

The **central processing unit** (CPU), or simply the **processor**, is the hardware unit that fetches program instructions from memory, decodes what they mean, and executes them. This three-step loop is called the **fetch–decode–execute** cycle.

The steps of the cycle are as follows:

1. **Fetch.** The processor takes the next instruction from memory. A special register keeps track of which instruction is next.
2. **Decode.** What the instruction asks for is determined: an addition, a comparison, a read from memory? Which values are to be used?
3. **Execute.** The requested operation is performed and the result is written to a register or to memory. Then the cycle starts again with the next instruction.

Three parts inside the processor run this cycle:

- The **arithmetic logic unit** (ALU) performs arithmetic operations (addition, subtraction) and logical operations (comparison, "and," "or").
- The **control unit** manages the order of the instructions and the flow of data. It decides which data goes to the ALU and where the result is written.
- **Registers** are very small, very fast storage cells inside the processor. The values the ALU is working on at that moment sit in the registers.

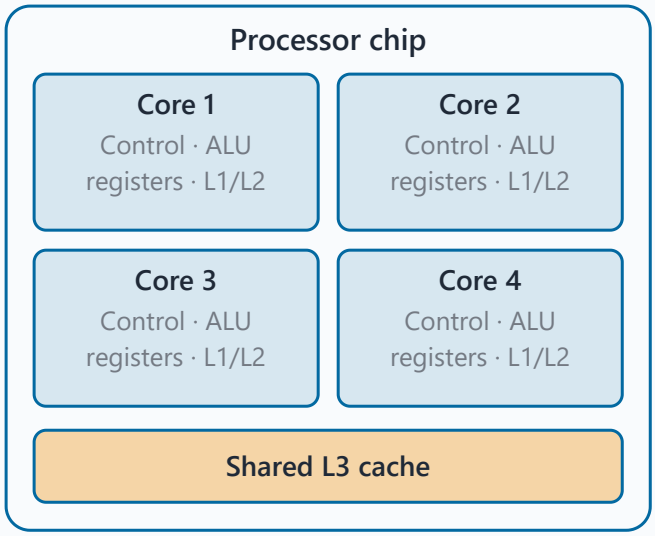


Figure 1.2.3: A four-core processor chip. Each core has its own control unit, ALU, registers, and L1/L2 cache. The L3 cache is shared by all cores.

Today's processors contain several **cores** on a single chip (Figure 1.2.3). Each core is a complete processing unit that can run the fetch–decode–execute cycle on its own. The **cache** (L1, L2, L3) in the figure is fast memory that keeps frequently used data right next to the processor; it is covered in detail in 1.2.4.

No single number determines a processor's performance. Four factors play a role together: the **number of cores**, the **clock speed** (GHz), the size of the **cache**, and the processor's **architecture**, that is, its internal design. Of two processors with the same clock speed, the newer-generation one can be noticeably faster because it does more work in each cycle.

Example 1.2.2

Memory holds the values `x = 5` and `y = 7`. The line `z = x + y` turns into roughly four simple instructions in the processor. Each instruction runs with a fetch–decode–execute cycle:

Instruction	Decode: what is asked?	Execute: result
1	read `x` from memory into register R1	R1 = 5
2	read `y` from memory into register R2	R2 = 7
3	add R1 and R2, write to R3 (ALU)	R3 = 12
4	write R3 to memory, in the place of `z`	z = 12

The ALU does the addition; in the read and write instructions, the control unit moves data between memory and the registers. A single line of addition takes four cycles. These steps can be followed one by one in the Simple CPU simulator.

1.2.2.2 Clock speed

Everything inside the processor proceeds to a rhythm. A **clock** signal sets this rhythm: the signal rises and falls at regular intervals, and at each rise the processor's circuits advance one step. Think of it as the conductor's beats in an orchestra; all the instruments play to the same beat.

Definition 1.2.3

Clock speed is the number of cycles the clock signal makes in one second, and it is measured in hertz (Hz). 1 GHz (gigahertz) is 10^9 cycles per second, that is, 1 billion cycles. The duration of one cycle is called a **clock cycle**:

> duration of one cycle = 1 ÷ clock speed

It is easy to see where this formula comes from. If 3 billion cycles are made per second, 1 second is divided into 3 billion equal parts; each part is 1/3,000,000,000 of a second. For such tiny durations the **nanosecond** (ns) is used: 1 ns = 10^{-9} s, one billionth of a second.

Example 1.2.3

How long does one clock cycle take on 3 GHz and 3.2 GHz processors? If the 3.2 GHz processor finished each instruction in 4 cycles on average, how many instructions would it execute per second?

1. **3 GHz.** 3 GHz = 3×10^9 cycles/s. One cycle: $1 \div (3 \times 10^9)$ s = $0.333\ldots \times 10^{-9}$ s ≈ **0.33 ns**.
2. **3.2 GHz.** $1 \div (3.2 \times 10^9)$ s = 0.3125×10^{-9} s = **0.3125 ns**.
3. **Number of instructions.** 3.2×10^9 cycles are made per second and each instruction takes 4 cycles: $3.2 \times 10^9 \div 4 = 0.8 \times 10^9$ = **800 million instructions/s**.

A comparison shows how short a cycle is: in 0.33 ns even light travels only about 10 cm.

A processor with a higher clock speed is not always faster. A processor's speed is determined by the clock speed multiplied by the amount of work done in each cycle. New architectures can advance several instructions at once in one cycle. So a new 3 GHz processor can be faster than an old 4 GHz one. Moreover, clock speed is not fixed: when the processor is idle it slows down to save energy, and under load it speeds up as far as temperature allows.

Common mistake

Comparing processors of different generations or different manufacturers by GHz alone. GHz is meaningful when comparing two processors in the same family. Because the work done per cycle differs across architectures, the comparison becomes misleading; measurements taken

on a real workload are more reliable.

1.2.2.3 Cores

Definition 1.2.4

A **core** is a unit inside the processor that can execute an independent stream of instructions. A multicore processor can run several instruction streams at the same time.

The sequence of instructions a program executes is called a **thread**. A single-threaded program executes its instructions in one line; it cannot split this line and distribute it to different cores. A program split into several threads, on the other hand, can use the cores at the same time. A video-conversion program can give different minutes of the film to different cores. Typing a letter in a text editor, however, is not big enough to be split.

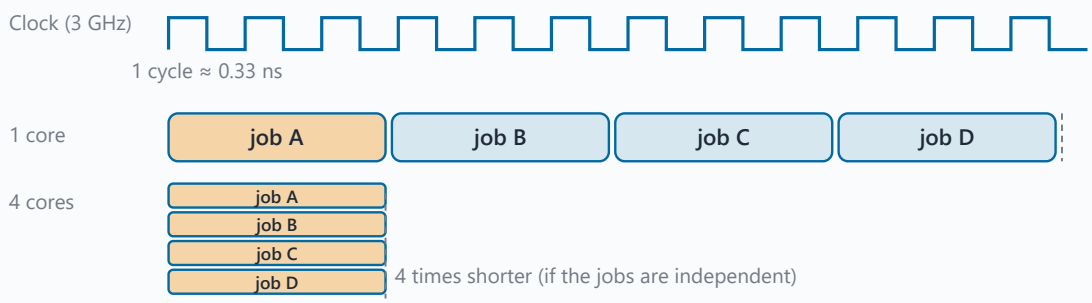


Figure 1.2.4: The clock signal and cores. At 3 GHz one cycle takes about 0.33 ns. Four independent jobs (A, B, C, D) run one after another on a single core and at the same time on four cores; if the jobs are independent, the total time falls to a quarter.

Figure 1.2.4 shows when many cores are useful: the jobs must be independent of one another. If the result of one job is needed as input by another, the second must wait for the first, and two cores cannot work together. Different programs running at the same time (a browser, a music player, an antivirus scanner), on the other hand, are naturally independent; the operating system distributes them across different cores.

Example 1.2.4

There are 7 independent jobs, each taking 5 seconds on a single core. The jobs cannot be split, and a core runs only one job at a time. What is the total time on a 3-core processor?

1. **First 5 seconds.** Three jobs are given to the three cores (1, 2, 3). Four jobs wait.
2. **Seconds 5–10.** The first three jobs finish and the next three (4, 5, 6) start. One job waits.

3. **Seconds 10–15.** Only job 7 is left. One core works and two sit idle.

The total time is **15 s**. On a single core it would have taken 7 × 5 = 35 s. The speedup is 35 ÷ 15 ≈ 2.3 times, not 3 times, because two cores are idle in the last round. The general rule: the number of rounds is the number of jobs divided by the number of cores, rounded up (7 ÷ 3 = 2.33 → 3 rounds).

Common mistake

Thinking "8 cores = 8 times the speed." A single-threaded program uses one core even on an 8-core processor and does not speed up. The number of cores makes a difference only if the work can be split or if many programs run at the same time.

1.2.2.4 Processor architectures and instruction sets

Each processor family understands a particular list of instructions. The complete set of instructions a processor understands is called its **instruction set**. The instruction set is the common language between the processor and software: two processors that understand the same instruction set can run the same machine code. Two instruction sets stand out today:

- **x86-64:** Common in desktop, laptop, and server computers. Intel and AMD processors use this instruction set.
- **ARM:** Common in phones, tablets, and systems where energy efficiency matters; it is increasingly used in laptops as well. Companies such as Apple, Qualcomm, and MediaTek design ARM-based processors.

RISC-V should be mentioned as a third option. RISC-V is an open instruction set: no license fee is paid to use it. It is therefore becoming more and more common, especially in embedded systems and in research.

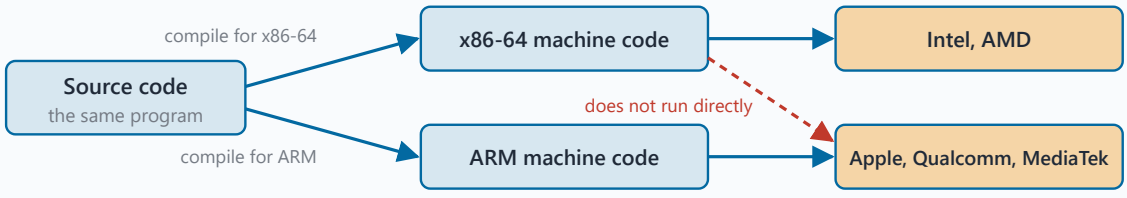


Figure 1.2.5: The same source code is compiled separately for two different instruction sets. x86-64 machine code runs on Intel and AMD processors, and ARM machine code on Apple, Qualcomm, and MediaTek processors. Code produced for one instruction set does not run directly on the other.

Figure 1.2.5 shows the difference between the source code a programmer writes and the machine code the processor runs. Source code is the form people read. A **compiler** translates this code into the machine code of a particular instruction set. If the same program is to run on two architectures, it is compiled twice. This is one reason a phone app does not run directly on a computer.

A brand or model name alone does not determine performance. Even if two processors come from the same manufacturer and have the same instruction set, their generations, power limits, and core designs can differ. A proper comparison is made on the workload to be used: office programs, games, and video processing benefit from different processor features.

1.2.2.5 SoC, mobile processors, and NPUs

In a desktop computer, the processor, the graphics card, and the other controllers are mostly separate parts. A phone has no room for that, and battery life matters a great deal. The solution is to put all the units on a single chip.

Definition 1.2.5

A **System-on-Chip** (SoC) is a design in which the CPU, GPU, memory controller, and other units are gathered on a single chip. A **neural processing unit** (NPU) is a unit designed to speed up artificial intelligence computations at low power.

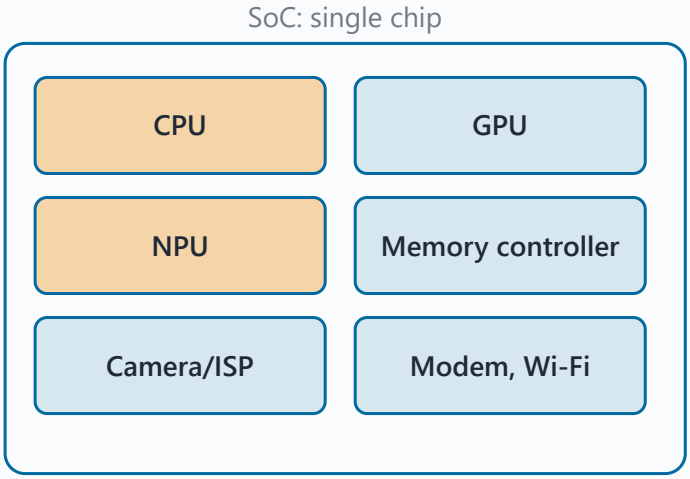


Figure 1.2.6: The components of a SoC: CPU, GPU, NPU, memory controller, camera image processor (ISP), and modem/Wi-Fi are on a single chip.

Having the units on one chip brings two gains. The first is saving space: a part small enough to fit inside a phone or a thin laptop. The second is saving power: data travels between units over short paths, and the energy spent on long connections between separate chips goes down. In most SoCs the

RAM is a separate chip, but it is placed very close to the SoC, even inside the same package. The details vary from manufacturer to manufacturer.

The jobs an NPU speeds up are ones we meet often in daily life: recognizing voice commands, reducing noise in a photo, face recognition, instant translation. These jobs contain many small multiplications and additions of the same kind. The CPU can do them too, but the NPU finishes the same job with much less energy.

The energy efficiency of mobile processors is sometimes attributed to the ARM instruction set alone. Yet the efficiency gap comes not only from the instruction set but also from the design: the structure of the cores, the manufacturing technology, and power management also determine the result. This is why ARM-based chips have also begun to be used in laptops.

1.2.3 Motherboard, RAM, and ROM

The processor cannot work alone. It reads its instructions and data from a memory, gets its electricity from the power supply, and sends its results to the graphics card and other units. The motherboard provides all these connections. This section covers the motherboard and two kinds of memory: RAM, which holds running programs, and ROM, where the firmware that manages the computer's startup lives.

1.2.3.1 The motherboard

Definition 1.2.6

The **motherboard** is the main circuit board that joins the components of the computer physically and electrically. The components communicate with one another through **buses** on the motherboard.

The motherboard can be compared to a city's road network. The processor is the city center; memory, storage, and the graphics card are different districts. The wider and smoother the roads, the faster traffic flows. On the motherboard there is a separate slot or connection point for each component (Figure 1.2.7).

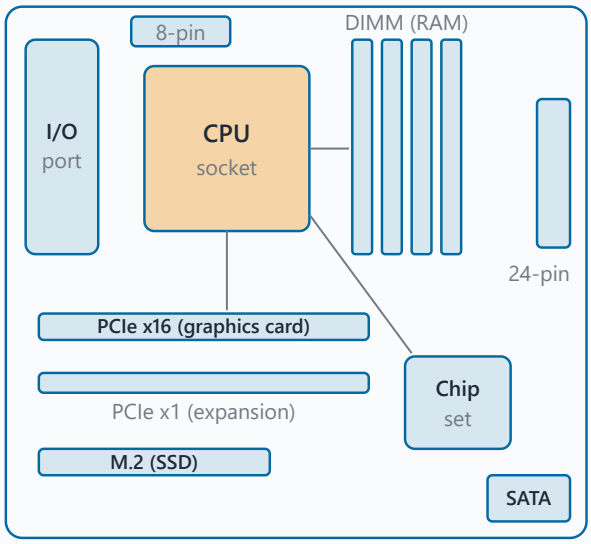


Figure 1.2.7: The layout of the motherboard: CPU socket, DIMM slots for RAM, PCIe x16 slot for the graphics card, PCIe x1 slot for expansion cards, M.2 slot for an SSD, SATA connections, chipset, power inputs (24-pin and 8-pin), and the I/O ports on the rear panel.

Part	Function
CPU socket	The slot the processor is plugged into. It determines the supported processor family.
DIMM slots	Hold the RAM modules.
PCIe slots	Graphics cards and other expansion cards are plugged in.
M.2 and SATA	Storage devices (SSD, HDD) are connected.
Chipset	Manages data traffic between ports and slots.
24-pin and 8-pin	Carry electricity from the power supply to the motherboard and the processor.

Table 1.2.2: Main connection points on the motherboard.

The abbreviations in the table often appear in the specifications of computer parts. **DIMM** (dual inline memory module) is the name of the RAM module in desktop computers. **PCIe** (PCI Express) is the high-speed bus to which devices such as graphics cards and fast SSDs connect; suffixes such as “x16” and “x4” show how many **lanes** are used. A lane is one of the parallel channels of the bus; the more lanes, the more data is carried at once. **M.2** is the form of slot into which small SSDs plug directly on the motherboard. **SATA** is the interface over which hard disks and older SSDs are connected by cable.

Another job of the motherboard is to determine compatibility. The CPU socket decides which processor can be fitted, the DIMM slots which generation of RAM can be used, and the chipset how many fast connections are supported at the same time. For this reason, when a computer is assembled part by part, the first decision is usually between the processor and the motherboard. Compatibility is treated as a whole in 1.2.10.1.

1.2.3.2 RAM

Definition 1.2.7

RAM (random access memory) is the main memory in which running programs and the data they use are held. It is **volatile**: its contents are lost when the power goes off. "Random access" means that every memory cell can be reached directly by its address, without waiting in line.

To understand the structure of RAM, it helps to think of it as a very long row of lockers. Each locker has a number, its **address**, and each locker holds exactly one **byte**. A byte consists of 8 bits and can hold an integer from 0 to 255. When the processor wants to read a piece of data, it states the address, and memory returns the byte at that address. To reach address 1000 there is no need to walk through the addresses from 0 to 999; every address is reached in the same time.

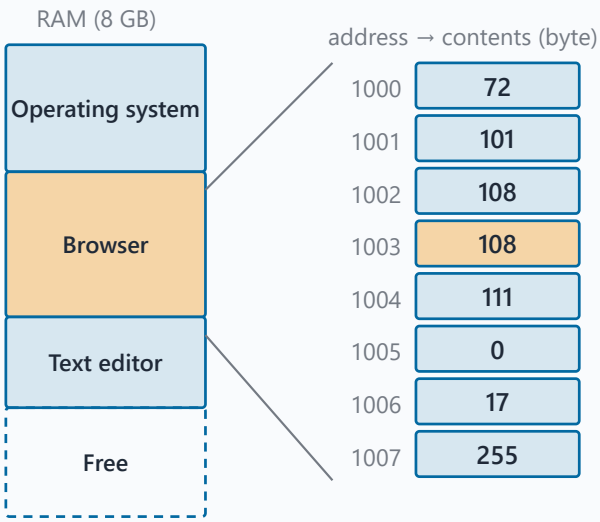


Figure 1.2.8: A moment in 8 GB of RAM. The operating system, the browser, and the text editor occupy separate regions of memory; part of it is free. Looking closer, every byte has an address: at address 1000 sits 72, at 1001 sits 101, at 1002 and 1003 sits 108, and at 1004 sits 111.

The numbers in Figure 1.2.8 were not chosen at random. Text is stored in memory as numbers; each character corresponds to a number. In the widely used ASCII encoding, 72 is "H," 101 is "e," 108 is "l," and 111 is "o." So the word "Hello" sits at addresses 1000–1004. Memory knows only numbers; whether these numbers are letters, colors, or sound samples is decided by the program that uses them.

Example 1.2.5

How does the value 72 at address 1000 sit in memory as 8 bits? How is the same value written in base 16?

1. **Binary.** The place values of 8 bits from left to right are 128, 64, 32, 16, 8, 4, 2, 1. Let us write 72 as a sum of these values: 72 = 64 + 8. The 64 and 8 positions become 1 and the others 0: **01001000**.
2. **Check.** 0·128 + 1·64 + 0·32 + 0·16 + 1·8 + 0·4 + 0·2 + 0·1 = 64 + 8 = 72.
3. **Hexadecimal.** The 8 bits are split into two groups of four: 0100 and 1000. Since 0100 = 4 and 1000 = 8, 72 is written **48** in base 16 (in programming often `0x48`).
4. **The largest value.** When all eight bits are 1, 11111111 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = **255**; its hexadecimal form is `0xFF`. The 255 at address 1007 in Figure 1.2.8 is therefore the largest value a byte can hold.

Four properties of RAM directly affect everyday use of a computer:

- **Volatility:** The contents are lost when the power goes off. This is why an unsaved document disappears in a power cut.
- **Fast access:** The CPU reaches every address directly. Access to RAM takes about 50–100 ns; this is thousands of times faster than storage devices.
- **Capacity:** The size of RAM determines how many programs can comfortably run at the same time. 8–64 GB is common in today's computers.
- **Compatibility:** The generation of the RAM module (DDR4, DDR5), its speed, and the motherboard's support must match.

What happens when RAM runs out? When there is no room left in RAM, the operating system writes some of the data that has not been used for a while to disk and frees the space. The area on disk used this way is called **swap space**, or the page file on Windows. When needed, the data is read back from disk. Because the disk is much slower than RAM, the computer slows down noticeably. Even if there is plenty of free space on the disk, nothing changes: the cause of the slowness is not lack of space but the speed of the disk.

Common mistake

Looking only at free disk space on a computer that has slowed down. Even if the storage device is almost empty, when RAM is insufficient the system uses the disk as temporary memory and programs slow down. In this case the remedy is not the disk but more RAM, or running fewer programs at the same time.

1.2.3.3 ROM and firmware

When a computer is turned on, RAM is empty. The operating system sits on the disk, but no software yet runs that can read the disk, start the screen, or recognize the keyboard. This "first step" problem is solved by a small piece of software that sits in a permanent memory on the motherboard.

Definition 1.2.8

ROM (read-only memory) is memory whose contents are not lost when the power goes off. **Firmware** is low-level software that manages the basic behavior of the hardware. In computers the firmware is called the **BIOS**, or **UEFI**, the newer standard that has replaced it.

Although the name ROM means "read-only," today firmware is mostly kept in rewritable **flash memory**. Flash memory also keeps its contents when the power goes off, but it can be rewritten when necessary. This allows the firmware to be updated.

The startup of a computer proceeds roughly in this order:

1. The power button is pressed; the processor starts running the UEFI that sits on the flash chip on the motherboard.
2. UEFI tests and initializes the hardware: RAM, graphics card, keyboard, and storage devices are recognized. This test is called the **power-on self-test** (POST).
3. UEFI scans the disks according to the **boot order** given in the settings and finds the program (the boot loader) that will start the operating system.
4. The boot loader loads the operating system from disk into RAM and hands control over to it.

A firmware update can be made to recognize a new processor, fix a compatibility problem, or close a security hole. It needs care, however: if the power goes off during the update, the software in the flash memory may be left half-written and the computer may fail to start. So an update is made only when necessary and with uninterrupted power.

Common mistake

Confusing ROM with RAM because their names are similar. RAM is volatile and holds the running programs; its size is measured in gigabytes. ROM (flash) is permanent and holds only the startup software; its size is a few tens of megabytes. The first software that runs when the computer is turned on sits not in RAM but in this permanent memory.

1.2.4 The memory hierarchy and the cache

An ideal memory would be very fast, very large, and cheap all at once. There is no such memory. Fast memory technologies are expensive and stay small; large, cheap memories are slow. Computers solve this problem by using several different memories together in layers.

1.2.4.1 The memory hierarchy

Definition 1.2.9

The **memory hierarchy** is the arrangement of the kinds of memory in a computer into layers by speed and capacity. The upper layers (registers, cache) are fast, small, and expensive; the lower layers (RAM, SSD, HDD) are slow, large, and cheap.

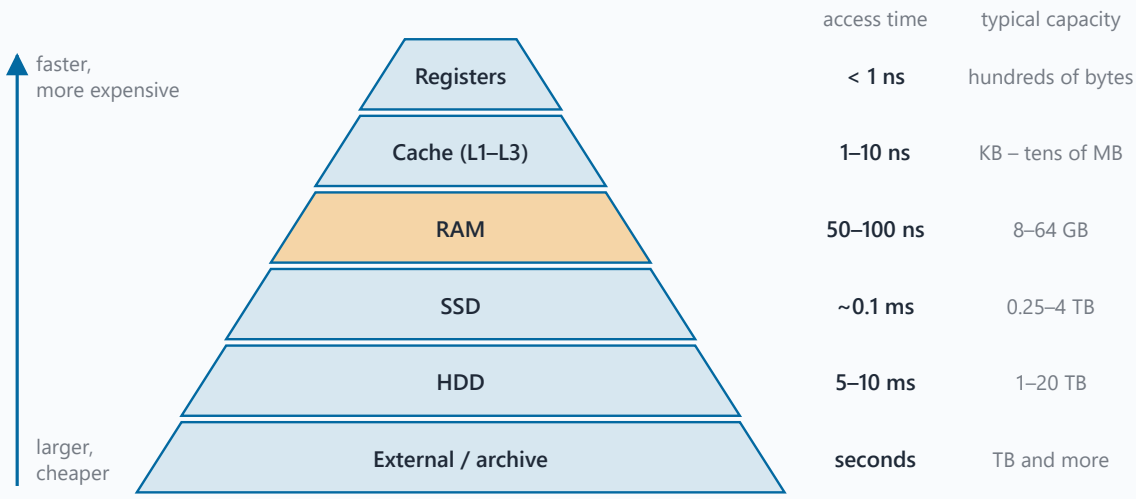


Figure 1.2.9: The memory hierarchy pyramid. Going up, access gets faster, capacity gets smaller, and the cost per byte rises. The values are approximate.

Layer	Access time	Typical capacity	Non-volatile?
Registers	< 1 ns	hundreds of bytes	no
Cache (L1–L3)	1–10 ns	KB – tens of MB	no
RAM	50–100 ns	8–64 GB	no
SSD	≈ 0.1 ms	0.25–4 TB	yes
HDD	5–10 ms	1–20 TB	yes
External / archive	seconds	TB and more	yes

Table 1.2.3: Approximate access times and typical capacities of the memory layers.

The hierarchy works because programs do not use memory in a disorderly way. Within a short time a program mostly works with the same small set of data: a loop reads the same variables again and again, and an array is walked through in order from start to end. If this small set is kept in the upper layers, the great majority of the processor's requests are served from the fast layer. The slow layers are visited only rarely.

The last column of the table also shows an important distinction. RAM and the layers above it are volatile; SSD and the layers below it are non-volatile. While working on a document, the changes sit in RAM; only the "Save" command writes them to the non-volatile layer.

1.2.4.2 Speed differences between layers

The differences between access times are so large that a special scale is needed to show them on the same line. Figure 1.2.10 uses a **logarithmic scale**: each grid line on the axis is 1,000 times the previous one (1 ns, 1 µs, 1 ms). That way nanoseconds and milliseconds fit in the same figure.

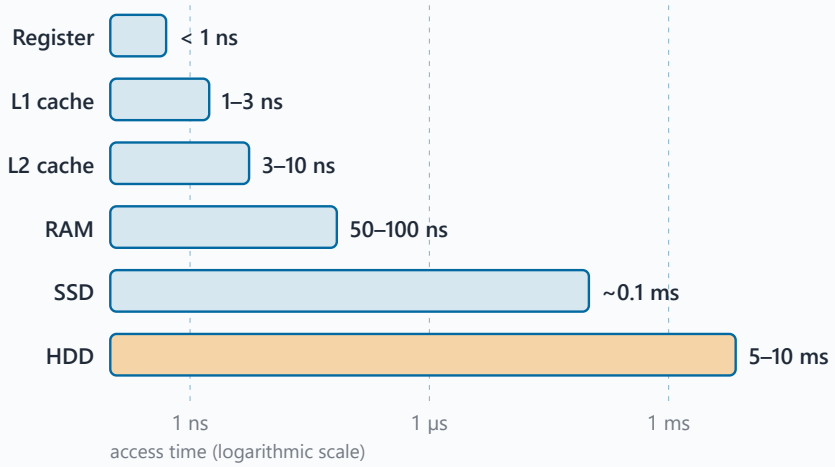


Figure 1.2.10: Access times of the memory layers on a logarithmic scale. A register takes less than 1 ns, the L1 cache 1–3 ns, the L2 cache 3–10 ns, RAM 50–100 ns, an SSD about 0.1 ms, and an HDD 5–10 ms.

Remembering how the units relate makes the calculations easier: 1 ms (millisecond) = 1,000 µs (microseconds) = 1,000,000 ns. So 0.1 ms is 100,000 ns.

Example 1.2.6

How many times slower is an SSD than RAM? How many times slower is an HDD than an SSD and than RAM?

1. **SSD versus RAM.** SSD ≈ 0.1 ms = 100,000 ns. RAM is 50–100 ns. 100,000 ÷ 100 = 1000, 100,000 ÷ 50 = 2000. An SSD is about **1000–2000 times** slower than RAM.
2. **HDD versus SSD.** An HDD is 5–10 ms. 5 ÷ 0.1 = 50, 10 ÷ 0.1 = 100. An HDD is **50–100 times** slower than an SSD.

3. **HDD versus RAM.** 5 ms = 5,000,000 ns and 10 ms = 10,000,000 ns. The smallest ratio is 5,000,000 ÷ 100 = 50,000 and the largest is 10,000,000 ÷ 50 = 200,000. An HDD is **50,000–200,000 times** slower than RAM.

Bringing this gap to an everyday scale builds intuition. If a read from RAM took 1 second, a read from an SSD would take about 17–33 minutes, and a read from an HDD between 14 hours and 2.3 days.

These ratios explain why the processor wants to read data not from disk but from RAM, and if possible from the cache. Even RAM is slow compared with one clock cycle of the processor: at 3 GHz a cycle is 0.33 ns, and waiting for a value from RAM takes about 150–300 cycles. The cache exists to reduce this waiting.

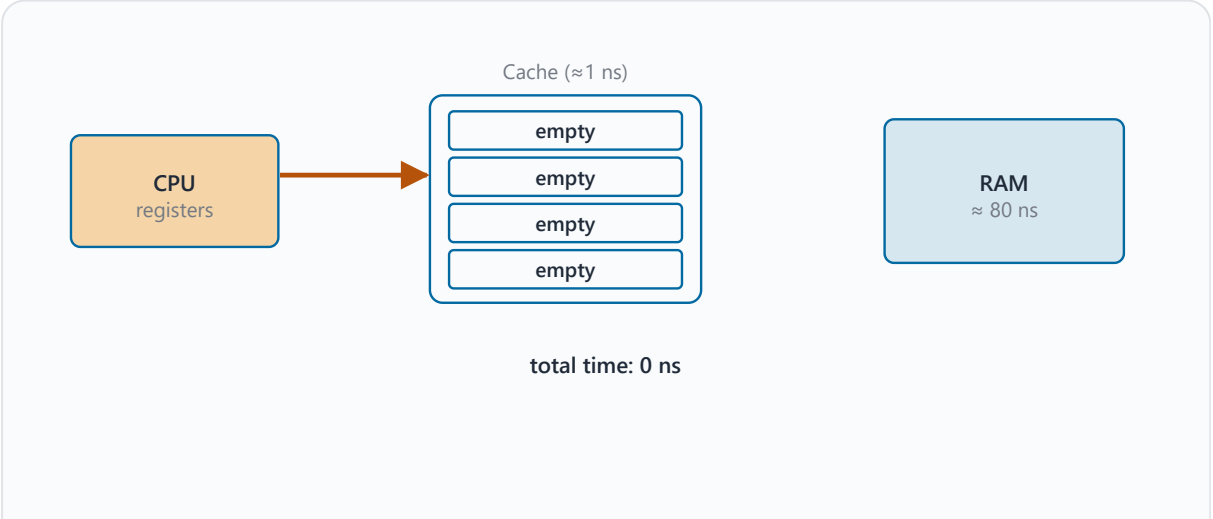
1.2.4.3 The cache: hits and misses

Definition 1.2.10

The **cache** is a small, fast memory that keeps copies of frequently used data next to the processor. If the data the processor asks for is found in the cache, this is called a **hit**; if it is not found, a **miss**. On a miss the data is fetched from RAM and a copy is written to the cache.

The working rule of the cache is simple. When the processor wants to read an address, it first looks in the cache. If the data is there, it is obtained in about 1 ns. If not, it goes to RAM; this takes about 80 ns. The data coming from RAM is not only handed to the processor but also copied into the cache. Moreover, the neighboring bytes next to the requested byte are brought along with it, because the next request is expected to be, in all likelihood, to a neighboring address.

Figure 1.2.11 shows this rule at four moments on the memory contents of Figure 1.2.8. In this example the cache is filled in blocks of two addresses: 1002–1003, 1004–1005, and so on.



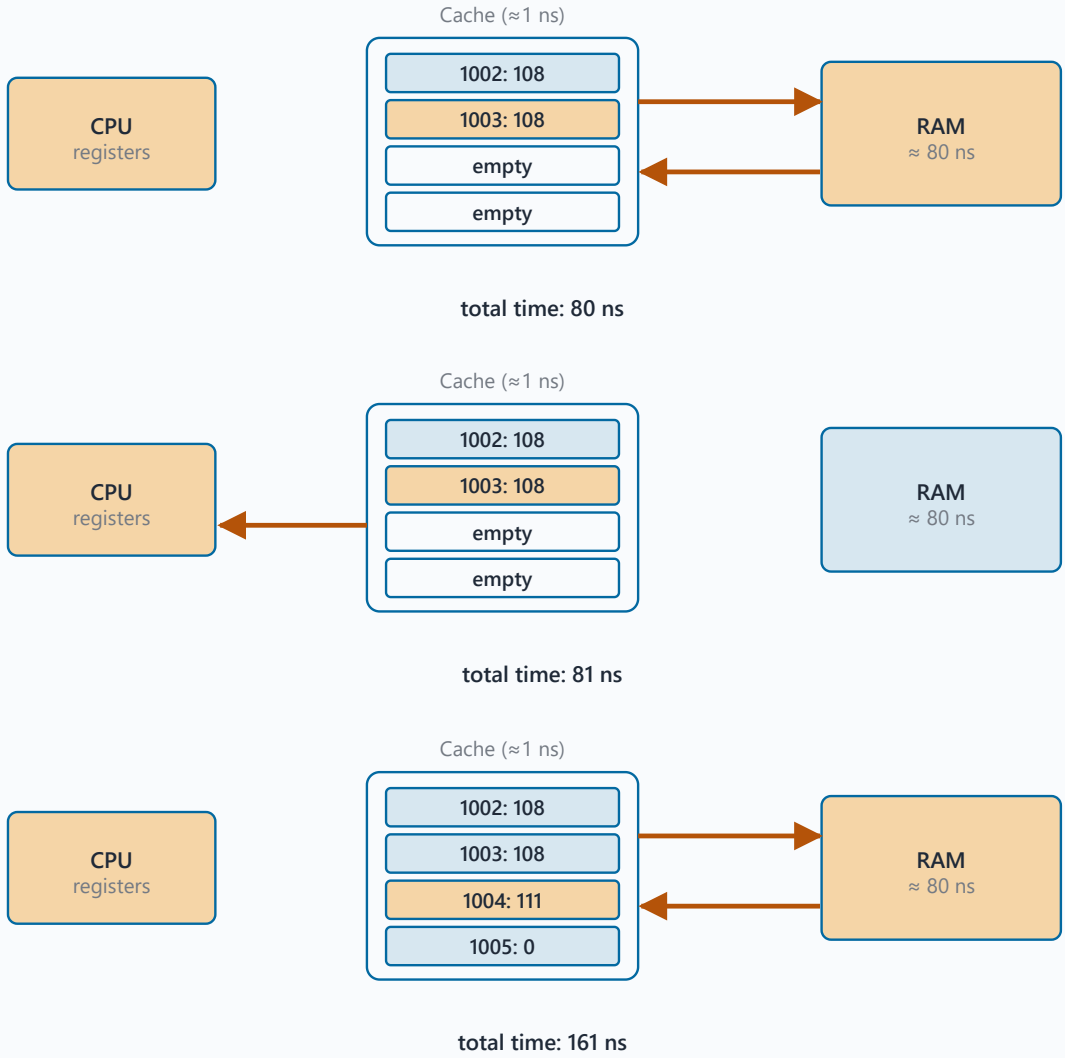


Figure 1.2.11: Hit and miss in the cache, four moments from top to bottom. (1) The cache is empty; the CPU asks for address 1003. (2) Miss: 1003 is fetched from RAM in 80 ns and written to the cache together with its neighbor 1002. (3) 1003 is requested again: hit, the value 108 arrives in about 1 ns; total 81 ns. (4) 1004 is requested: it is not in the cache, a miss; the block 1004–1005 is fetched from RAM, and the total becomes 161 ns.

Request	In the cache?	Result	Time	Total	Cache after the request
1003 (first)	no	miss	80 ns	80 ns	1002, 1003
1003 (again)	yes	hit	1 ns	81 ns	1002, 1003
1004	no	miss	80 ns	161 ns	1002, 1003, 1004, 1005

Table 1.2.4: Trace of the three requests in Figure 1.2.11 (hit ≈ 1 ns, miss ≈ 80 ns).

The three requests took 161 ns with the cache. Without the cache each request would go to RAM and the total would be 3 × 80 = 240 ns. The gain comes from the second request: because the same address was requested again within a short time, there was no need to go to RAM.

Real caches are much larger than this example, but they are limited. When the cache is full, room must be made for a new block. Most caches evict the block that has gone unused for the longest time. In these small examples the question of space is ignored.

1.2.4.4 The cache in an array sum: spatial locality

The cache's bringing neighboring bytes along relies on a very common behavior in programs. When an array (a structure in which values of the same type are laid out one after another in memory) is walked from start to end, each element sits right next to the previous one. Let us count how much time this behavior saves on the sum of an eight-element array. The program is:

```
sum = 0
for each i = 0, 1, …, 7
    sum = sum + a[i]
// 4-byte integers, 16-byte line
```

The calculation uses these assumptions: an integer takes 4 bytes; each time, a 16-byte block is brought from RAM into the cache. This block is called a **cache line**. Since 16 ÷ 4 = 4, four elements fit in one line: a[0]…a[3] are in the first block and a[4]…a[7] in the second. A hit takes about 1 ns and a miss about 80 ns. The values of the array are 3, 1, 4, 1, 5, 9, 2, 6.

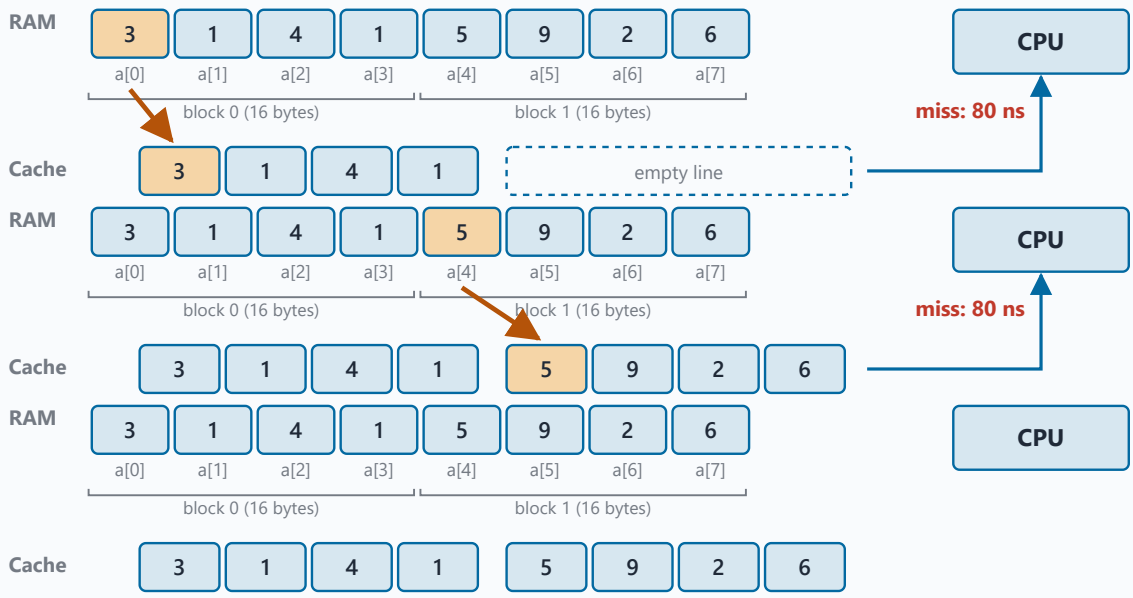


Figure 1.2.12: The cache in an array sum. At the top, a[0] has been requested and has missed; the whole of block 0 (a[0]…a[3]) has come into the cache. In the middle, a[4] has been requested and on the second miss block 1 (a[4]…a[7]) has come. At the bottom the loop has finished: both blocks are in the cache.

`i`	a[i]	Block	Result	sum	hit	miss	Time
0	3	0	miss	3	0	1	80 ns

`i`	a[i]	Block	Result	sum	hit	miss	Time
1	1	0	hit	4	1	1	81 ns
2	4	0	hit	8	2	1	82 ns
3	1	0	hit	9	3	1	83 ns
4	5	1	miss	14	3	2	163 ns
5	9	1	hit	23	4	2	164 ns
6	2	1	hit	25	5	2	165 ns
7	6	1	hit	31	6	2	166 ns

Table 1.2.5: Trace of the array sum. Each row shows the state after one loop step.

Table 1.2.5 gives the result clearly: the sum is 31; of the 8 accesses, 6 are hits and 2 are misses. The time is 2 × 80 + 6 × 1 = **166 ns**. Without the cache every access would go to RAM and take 8 × 80 = **640 ns**. The cache has sped this loop up about 3.9 times. The average time per access has also fallen to 166 ÷ 8 ≈ 20.8 ns.

Definition 1.2.11

Spatial locality is the tendency of a program that accesses one address to access nearby addresses soon afterward. The cache's fetching a whole line instead of only the requested byte exploits this tendency.

Example 1.2.7

What would the times be if the same array had 16 elements and were walked in two different ways? (a) The elements are read in order. (b) Only a[0], a[4], a[8], a[12] are read.

1. **(a) In order.** The 16 elements spread over 4 blocks. The first element of each block misses and the remaining three hit: 4 misses, 12 hits. The time is 4 × 80 + 12 × 1 = 332 ns. Without a cache it would be 16 × 80 = 1280 ns.
2. **(b) Skipping by four.** Each element read is in a different block, and each block is used only once. All 4 of the 4 accesses miss: 4 × 80 = 320 ns. The number of elements read has fallen to a quarter, but the time has hardly shortened at all.

When skipping, the neighboring elements the cache brings in are never used. The time per access is 332 ÷ 16 ≈ 20.8 ns in (a) but 80 ns in (b).

Common mistake

Thinking that as the number of elements read goes down, the time will shrink in the same proportion. In a system with a cache, what mainly determines the time is how many times RAM is visited. A few accesses to scattered addresses can take longer than many accesses to

consecutive addresses.

1.2.4.5 Multilevel caches: temporal locality

Real processors have several levels instead of a single cache. **L1** is the smallest and fastest level and sits inside each core. **L2** is larger but a little slower. **L3** is the largest and slowest cache level; it is usually shared by all cores (Figure 1.2.3). The processor looks for a piece of data starting from the fastest level:

```
In L1?  Yes → read, done
In L2?  Yes → copy to L1, read
In L3?  Yes → copy to L2 and L1, read
Fetch from RAM → copy to L3, L2, L1, read
```

The search spends some time at each level; if the data is not found, it moves to the next level down. So the cost of a miss is the sum of the times of all the levels searched. In the example below, L1 ≈ 1 ns, L2 ≈ 3 ns, L3 ≈ 10 ns, and RAM ≈ 80 ns are assumed. At the start, L1 holds B, L2 holds C, and L3 holds B, C, and D; A is only in RAM. The processor will read addresses A, A, and D in turn.

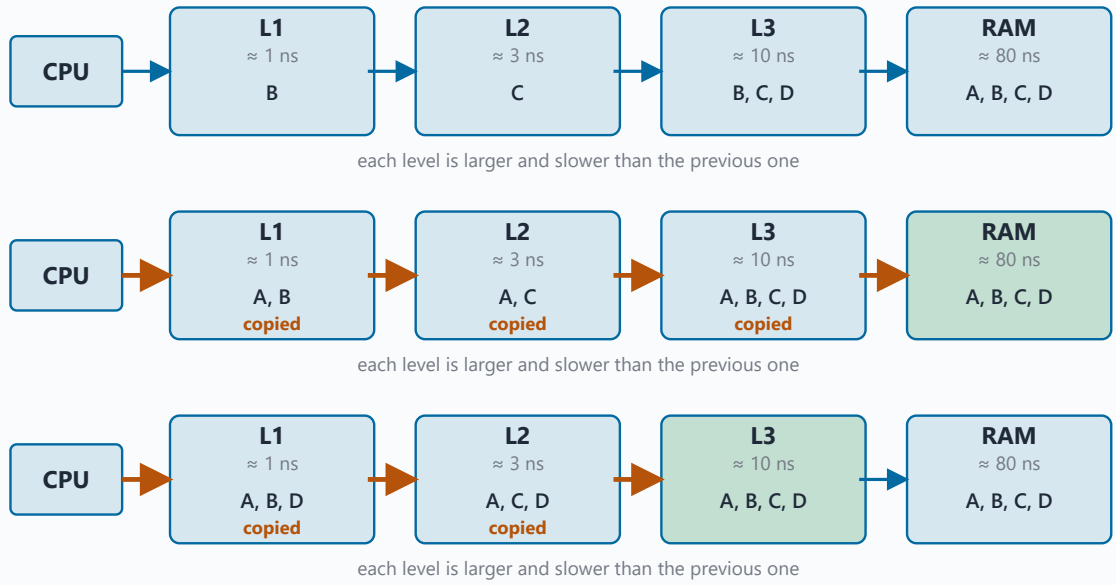


Figure 1.2.13: A read in a multilevel cache. Top, the initial state. Middle, the state after A has been fetched from RAM: A has been copied to all levels. Bottom, the state after D has been found in L3: D has been copied to L2 and L1.

Request	Search path	Time	Then L1 / L2 / L3
A	not in L1 (1) → not in L2 (+3) → not in L3 (+10) → RAM (+80)	94 ns	A, B / A, C / A, B, C, D
A	in L1	1 ns	unchanged

Request	Search path	Time	Then L1 / L2 / L3
D	not in L1 (1) → not in L2 (+3) → in L3 (+10)	14 ns	A, B, D / A, C, D / A, B, C, D

Table 1.2.6: Trace of the three reads.

The three reads took 94 + 1 + 14 = **109 ns** in all. If all three had been read directly from RAM, it would have been 3 × 80 = **240 ns**. Two details in the table are important:

- The first read of A is longer than an uncached read (80 ns): 94 ns. The price of a miss is the time of the levels searched in vain. A cache does not speed up every single read; the gain comes from reusing the same data.
- D was found in L3: 14 ns. Going to RAM would have taken 94 ns; 94 ÷ 14 ≈ 6.7, so it is about 7 times faster.

Definition 1.2.12

Temporal locality is the tendency of a program that accesses a piece of data to access the same data again within a short time. The second read of A taking 1 ns results from this tendency.

Spatial and temporal locality together explain why the cache is so effective. A loop's counter variable is read again in every round (temporal locality); the array the loop walks through sits at consecutive addresses (spatial locality). In typical programs the great majority of the processor's requests are served from the cache.

Example 1.2.8

95% of a program's memory requests hit in the cache. A hit takes 1 ns, and a miss takes 81 ns, the sum of looking in the cache (1 ns) and fetching from RAM (80 ns). What is the average time of a request?

1. Of 100 requests, 95 hit: 95 × 1 = 95 ns.
2. The remaining 5 miss: 5 × 81 = 405 ns.
3. The total is 95 + 405 = 500 ns; the average per request is 500 ÷ 100 = **5 ns**.

Without a cache each request would take 80 ns. Even a 5% miss rate dominates the time: more than 4 of the 5 ns come from misses. A small increase in the hit rate therefore brings a big gain.

1.2.5 Graphics card, connections, and sound

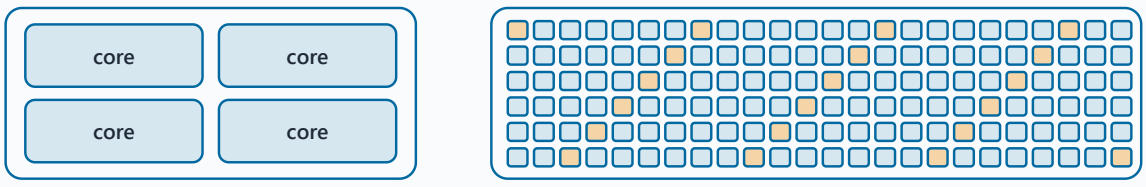
The processor computes the result, but other units are needed to turn this result into an image on the screen or a sound from the speaker. This section covers the graphics processing unit that produces the image, the ports that connect devices to the computer, and the sound card that converts sound to numbers and numbers to sound.

1.2.5.1 The graphics processing unit

Definition 1.2.13

A **graphics processing unit** (GPU) is a processor designed to carry out many similar operations at the same time. It takes its name from the job of computing the colors of millions of pixels. The card that carries the GPU is also called the **graphics card**.

Producing an image on a screen means repeating the same calculation for a large number of pixels. A screen with a resolution of 1920 × 1080 has 1920 × 1080 = 2,073,600 pixels. If 60 frames are to be shown per second, more than 124 million pixel colors must be computed every second. Each of these calculations is simple and independent of the others. For such a job, thousands of simple cores suit better than a few powerful ones.



CPU: a few powerful cores

GPU: thousands of simple cores do the same operation in parallel

Figure 1.2.14: A CPU consists of a few powerful cores; in a GPU, thousands of simple cores do the same operation in parallel.

Figure 1.2.14 summarizes the difference between the two kinds of processor. A CPU's cores are suited to carrying out complex and varied jobs quickly: make a decision, open a file, finish one calculation and move to another job. A GPU's cores are individually weak, but thousands of them execute the same instruction on different data at the same time. It is like everyone in a classroom solving their own problem at once: much faster than a single teacher solving all the problems one after another.

GPUs come in two forms:

- An **integrated GPU** is in the same package as the processor and shares the system memory (RAM). It uses little power and adds no extra cost; it is enough for office work, watching video, and light games.

- A **discrete GPU** is a separate card. It has its own processor and its own **video memory** (VRAM). It is more powerful, but uses more power and produces more heat.

The parallel computing power of GPUs is no longer limited to graphics. Video processing, scientific computing, and especially the training of artificial intelligence models also involve many independent multiplications and additions of the same kind. When choosing a GPU, the workload, the amount of VRAM, power consumption, cooling, and whether the software to be used supports that GPU are considered together.

> **Common mistake**
>
> Thinking that a GPU can take the place of a CPU. A GPU is fast only at many jobs of the same kind that can be split up. Jobs such as running the operating system or carrying out a program's step-by-step decisions stay on the CPU. The two processors complement each other.

1.2.5.2 Video connections

The image the graphics card produces is carried to the monitor by a cable. The type of cable and port determines the resolution and refresh rate that can be carried, and whether sound goes over the same cable too.

Connection	Signal carried	Common use
HDMI	digital video and audio	monitors, televisions, projectors
DisplayPort	digital video and audio	computer monitors
USB-C	video, data, and power depending on the model	laptops and mobile devices
VGA	analog video	older screens and projectors

Table 1.2.7: Common video connections.

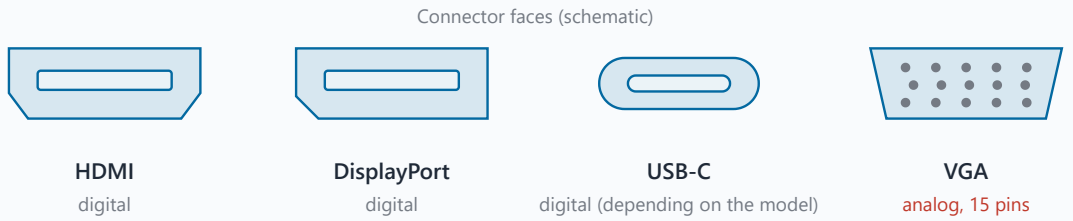


Figure 1.2.15: The faces of video connectors (schematic). HDMI has a trapezoid with cut lower corners; DisplayPort has only one corner cut; USB-C is oval and symmetric; VGA is an analog connector with 15 pins.

The difference between analog and digital signals matters here. A **digital** connection carries the color of each pixel as a number; the receiver reads these numbers as they are. An **analog** connection (VGA) carries colors as continuously varying voltages. As the cable gets longer or electrical noise around it

grows, these voltages get distorted and blur and ghosting appear in the image. For this reason VGA is used today only for compatibility with older devices.

The shapes in Figure 1.2.15 also explain why connectors cannot be plugged into the wrong port. The cut corners ensure that the connector fits in only one orientation. USB-C, being symmetric both ways, has no wrong-way problem.

1.2.5.3 USB and other ports

USB (Universal Serial Bus) is the common interface to which many devices, from keyboards to external disks, connect. The key to understanding USB is to tell two separate concepts apart: the **connector shape** and the **version**.

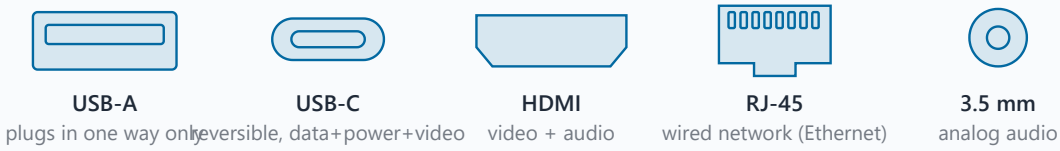


Figure 1.2.16: Common ports. USB-A plugs in one way only; USB-C is reversible and can carry data, power, and video; HDMI carries video and audio; RJ-45 is the wired network (Ethernet) port; the 3.5 mm jack carries analog audio.

The connector shape is the shape of the plug: type A (wide, rectangular) or type C (small, oval). The version is the standard that determines the speed at which data is carried. Table 1.2.8 shows the main versions and their top speeds.

USB version	Top speed	Typical use
USB 2.0	480 Mbit/s	keyboard, mouse
USB 3.2 Gen 1 (formerly USB 3.0)	5 Gbit/s	external disk
USB 3.2 Gen 2	10 Gbit/s	fast external SSD
USB4	20 or 40 Gbit/s	USB-C; display, docking station
Thunderbolt 4	40 Gbit/s	USB-C; display, docking station

Table 1.2.8: USB versions and their highest theoretical speeds.

The speeds in the table are in **bits**. File sizes, on the other hand, are mostly given in **bytes**. In units, a lowercase “b” stands for bit and an uppercase “B” for byte: 480 Mbit/s and 480 MB/s are very different speeds. Because a byte is 8 bits, a speed in bits is converted to bytes by dividing by 8.

Example 1.2.9

A 5 GB (5000 MB) file is copied at the highest theoretical speed. How many seconds does it take with USB 2.0 and with USB 3.2 Gen 2?

1. **USB 2.0.** 480 Mbit/s ÷ 8 = 60 MB/s. Time = size ÷ speed = 5000 ÷ 60 ≈ **83.3 s**.
2. **USB 3.2 Gen 2.** 10 Gbit/s = 10,000 Mbit/s; ÷ 8 = 1250 MB/s. Time = 5000 ÷ 1250 = **4 s**.

For the same file the difference is about 21 times. Real speeds are lower than the theoretical values, because the protocol's own control data, the speed of the disk, and the quality of the cable also affect the result.

Common mistake

Thinking that every USB-C port has the same speed and can output video. USB-C is only a connector shape. One USB-C port on a laptop may run at USB 2.0 speed (480 Mbit/s) while another is USB4 (40 Gbit/s). Video output and charging support also vary by port; this information is in the device's specifications or in the symbols next to the port.

1.2.5.4 The sound card

Sound is pressure waves in the air. A microphone turns these waves into a continuously varying electric voltage; this is called an **analog signal**. A computer, however, works only with numbers. The job of the sound card is to translate between these two worlds.

Definition 1.2.14

The **ADC** (analog-to-digital converter) measures the analog signal at regular intervals and turns each measurement into a number. Each measurement is called a **sample**, and the number of measurements per second is the **sampling rate**. The **DAC** (digital-to-analog converter) turns the numbers back into the electric signal that drives the speaker.

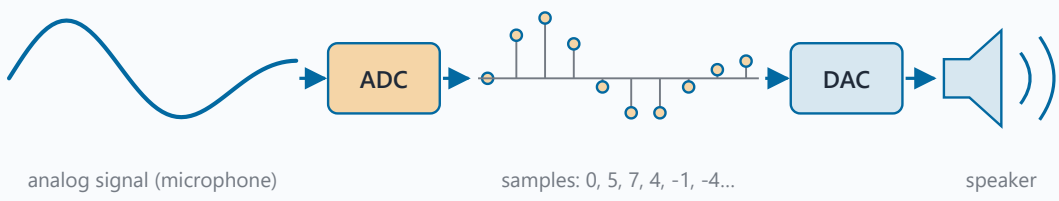


Figure 1.2.17: The analog signal from the microphone is sampled in the ADC and turned into numbers (0, 5, 7, 4, −1, −4, …). During playback the DAC turns these numbers back into a signal, and the speaker produces the sound.

The higher the sampling rate, the more accurately the sound is recorded. On music CDs the sampling rate is 44,100 samples per second (44.1 kHz). Each sample is stored in 16 bits, that is, 2 bytes. A stereo recording has two separate channels, left and right.

Example 1.2.10

A sound recording is made as a single channel (mono), at 48,000 samples/s and 16 bits per sample. How many bytes does a 30-second uncompressed recording take? How many MiB is that?

1. **One second.** 48,000 samples × 2 bytes × 1 channel = 96,000 bytes.
2. **30 seconds.** 96,000 × 30 = **2,880,000 bytes**.
3. **Converting to MiB.** 1 MiB = 1024 × 1024 = 1,048,576 bytes. 2,880,000 ÷ 1,048,576 ≈ **2.75 MiB**.

If the same recording were stereo, the size would double. Compression formats such as MP3 shrink this size about tenfold by discarding details the ear does not notice.

Most motherboards include an audio unit that is sufficient for everyday use: the headphone and microphone jacks are on the rear panel of the motherboard or on the front of the case. Where recording quality matters, when several microphones must be connected, or for professional music work, an **external sound card** is used. External cards contain a better ADC and DAC; they also sit away from the electrical noise inside the computer.

1.2.6 Case, cooling, and power supply

However powerful a computer's components are, it will not work efficiently if it is not properly powered and cooled. The case and the power supply are the parts most likely to be overlooked, yet they directly affect the stability of the system.

1.2.6.1 Case and cooling

Every component inside the computer turns almost all of the electrical energy it uses into heat. A processor drawing 125 W behaves like a small 125 W heater. If this heat is not carried away, the temperature rises quickly.

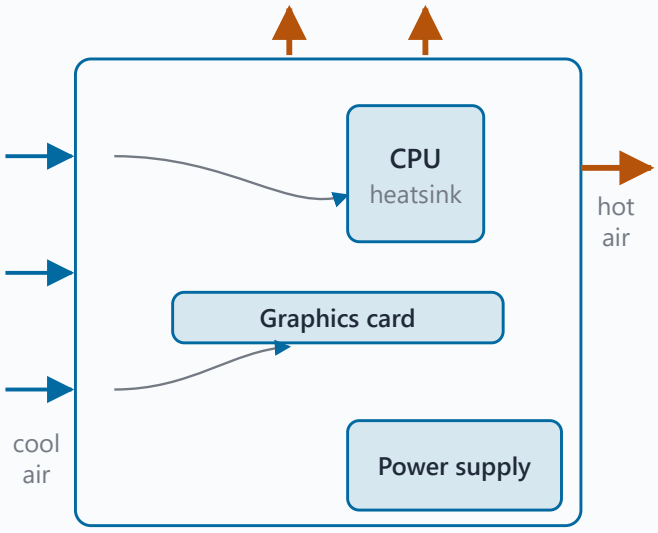


Figure 1.2.18: Airflow inside the case. Cool air enters at the front, passes over the processor heatsink and the graphics card; the warmed air leaves at the back and the top.

The case has four jobs:

- **Protection:** It protects the components from impact, dust, and static electricity that can come from touch.
- **Airflow:** Cool air enters at the front and hot air leaves at the back or the top (Figure 1.2.18). Because hot air rises, an exit at the top supports the natural flow.
- **Form factor:** The size and internal layout of the case determine which motherboard and power supply sizes will fit. Common motherboard sizes are ATX and the smaller mATX.
- **Cable layout:** Bundling the cables does not obstruct airflow and makes maintenance easier.

A **heatsink** is placed on top of the processor. The heatsink takes heat from the processor and spreads it over metal fins with a large surface; a fan blows air between the fins. In more powerful systems liquid cooling is used: the heat is transferred to a liquid, and the liquid is cooled in a radiator with fans.

Definition 1.2.15

Thermal throttling is the processor's automatic lowering of its clock speed when its temperature approaches the safe limit. This reduces the heat produced and protects the chip, but performance falls too.

Thermal throttling is a protective mechanism; it prevents the processor from being damaged. But if it happens often, it shows that the cooling is insufficient. This is one of the common reasons a dust-clogged laptop slows down over time. High temperature can also shorten the life of components in the long run.

1.2.6.2 The power supply

Definition 1.2.16

The **power supply unit** (PSU) is the unit that converts the **alternating current** (AC) of the mains into the **direct current** (DC) voltages the computer components use.

The electricity from the wall socket is, in Türkiye, alternating current at 230 V and 50 Hz: the direction of the voltage reverses back and forth 50 times a second. The computer's circuits, on the other hand, need steady, low voltages whose direction does not change. The power supply makes this conversion and produces different voltages for different components (Figure 1.2.19).

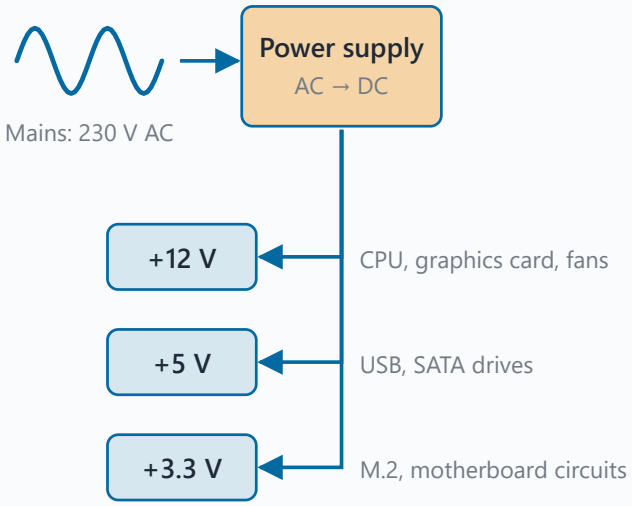


Figure 1.2.19: The power supply converts the 230 V AC of the mains into three DC voltages: +12 V powers the processor, the graphics card, and the fans; +5 V powers USB and SATA drives; +3.3 V powers M.2 devices and the motherboard circuits.

Four criteria are considered when choosing a power supply:

- **Capacity (W):** The highest power the supply can deliver. It must meet the total need of the components with a safe margin.
- **Efficiency:** How much of the power drawn from the mains reaches the components. The rest turns into heat.
- **Connections:** The motherboard (24-pin), processor (8-pin), graphics card, and storage cables must suit the system.
- **Protection:** In case of overcurrent, overvoltage, and short circuit, the power supply must protect itself and the components.

Example 1.2.11

In a system the processor draws 125 W, the graphics card 220 W, and the other components 75 W. A 30% safety margin is to be left. Models of 450, 550, 650, and 750 W are on the market. Which is chosen?

1. **Total need.** 125 + 220 + 75 = 420 W.
2. **Need with the margin added.** A 30% margin means multiplying the need by 1.3: 420 × 1.3 = 546 W.
3. **Choosing the model.** The smallest model that is not below 546 W is **550 W**.

There are two reasons for leaving a margin. Components can draw more than the stated power for brief moments. Also, power supplies get hotter and less efficient when they run very close to their limits.

Example 1.2.12

A power supply with 90% efficiency delivers 500 W to the components. How much power does it draw from the mains? How much turns into heat?

1. Since efficiency = delivered power ÷ drawn power, the drawn power = 500 ÷ 0.90 ≈ **555.6 W**.
2. The part that turns into heat = 555.6 − 500 ≈ **55.6 W**.

At the same load, an 80%-efficient power supply would draw 500 ÷ 0.80 = 625 W; the part turning into heat would be 125 W. An efficient power supply reduces both the electricity bill and the heat inside the case.

Common mistake

Taking a power supply's watt rating to be the power the computer draws continuously. A 650 W power supply draws around 200 W if the system asks for only 200 W. The watt rating is an upper limit, not continuous consumption.

1.2.7 Input devices

A computer receives information from the outside world through input devices. The keyboard and the mouse are the most familiar examples; but a phone's location sensor, a smartwatch's heart-rate monitor, or a temperature sensor in a factory are input devices too.

1.2.7.1 Kinds of input devices

Definition 1.2.17

An **input device** is a device that converts information from the user or the environment into digital data the computer can process and passes it to the system.

Device	Information taken in
Keyboard	text, numbers, and commands
Mouse, touchpad	pointing, selecting, and movement
Microphone	sound
Camera, scanner	images and documents
Game controller	axis, button, and motion information
Sensor	measurement of temperature, position, light, or motion

Table 1.2.9: Main input devices and the information they take in.

All input devices share one step: a physical quantity is turned into a number. A key press on the keyboard becomes a key code, sound at the microphone becomes samples (1.2.5.4), and light at the camera becomes three color values for each pixel. The computer processes these numbers; it does not know they were once sound or light.

Note the word “scanner.” Here it means the device that turns a paper document into an image. A web browser is also a program that can be used to read pages on the internet, but it is software, not an input device.

1.2.7.2 The keyboard

The keyboard is the basic tool for entering text. Its keys are divided into regions by function (Figure 1.2.20):

- **Alphanumeric keys** enter letters, digits, and punctuation marks.
- **Function keys** (F1–F12) take on special tasks depending on the application; for example, in many programs F1 opens help.
- **Modifier keys** (Ctrl, Alt, Shift, the Windows key) type nothing on their own; pressed together with other keys they form **shortcuts**.
- **Arrow keys** move the cursor; the **numeric keypad** makes it easier to enter many digits.

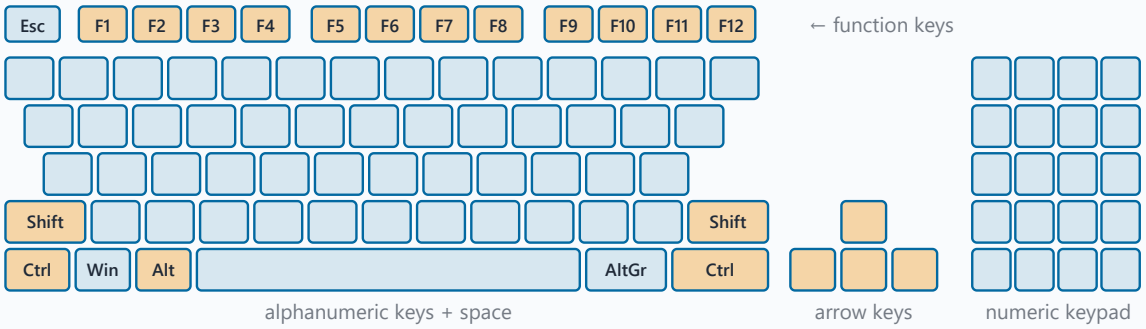


Figure 1.2.20: The regions of the keyboard: function keys at the top, the alphanumeric section and modifier keys (Shift, Ctrl, Win, Alt, AltGr) in the middle, arrow keys and the numeric keypad on the right.

When editing text, the most frequently used keys are Enter, Backspace, and Delete. Enter confirms an entry or moves to a new line. **Backspace** deletes the character *to the left* of the cursor, and **Delete** deletes the character *to the right* of the cursor. If some text is selected, both delete the whole selection. Frequently used shortcuts are these:

Shortcut	Function
Ctrl + C	Copy
Ctrl + X	Cut
Ctrl + V	Paste
Ctrl + Z	Undo
Alt + Tab	Switch between open applications

Table 1.2.10: Common keyboard shortcuts.

The copy and cut commands put the selected text into a temporary area called the **clipboard**. Paste inserts the clipboard's contents where the cursor is; the clipboard does not change in the process. So the same content can be pasted many times in a row.

Example 1.2.13

A text box contains "Ders" (Turkish for "lesson") and the cursor is at the end of the text. The clipboard is empty. The following operations are performed in order: select all the text and press Ctrl + C; clear the selection and put the cursor at the end; Ctrl + V; Backspace; Ctrl + Z. What are the text and the clipboard after each step? (The position of the cursor is shown with `|`.)

Step	Operation	Text	Clipboard
0	start	`Ders|`	empty
1	select all, Ctrl + C	`Ders` (selected)	`Ders`
2	clear the selection, cursor to the end	`Ders|`	`Ders`

Step	Operation	Text	Clipboard
3	Ctrl + V	`DersDers|`	`Ders`
4	Backspace	`DersDer|`	`Ders`
5	Ctrl + Z	`DersDers|`	`Ders`

Ctrl + C does not change the text; it only fills the clipboard. Backspace deletes the “s” to the left of the cursor. Ctrl + Z undoes the last operation, that is, the deletion. The clipboard stays “Ders” throughout.

Common mistake

Thinking that Delete and Backspace delete the same character. When the cursor is at the end of the text, Delete deletes nothing, because there is no character to the right of the cursor. Backspace deletes the last character.

1.2.7.3 The mouse and pointing devices

The mouse is a **pointing device** used to steer the cursor on the screen. The optical sensor underneath photographs the surface thousands of times per second and, by comparing consecutive images, calculates how far and in which direction it has moved. Two numbers stand out in a mouse’s technical specifications.

Definition 1.2.18

DPI (dots per inch) is the number of steps the mouse reports to the computer when it moves 1 inch (2.54 cm). The **polling rate** is how many times per second the mouse sends its position, and it is measured in Hz.

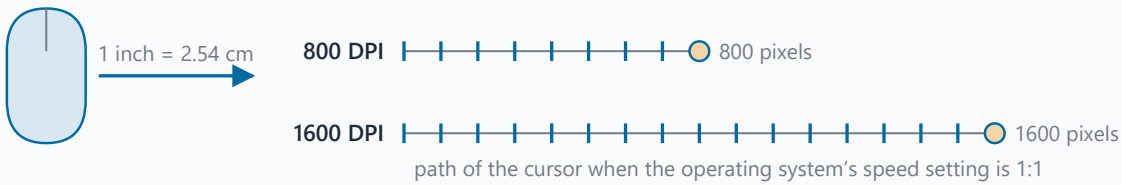


Figure 1.2.21: When the mouse moves 1 inch, an 800 DPI mouse reports 800 steps and a 1600 DPI mouse reports 1600. With the operating system’s speed setting at 1:1, the cursor moves that many pixels.

Example 1.2.14

A 1600 DPI mouse is slid 3 inches (about 7.6 cm) on the desk. The speed setting is 1:1. How many pixels does the cursor move? If the mouse's polling rate is 500 Hz, how often, in ms, is the position sent?

1. **Number of steps.** 1600 steps/inch × 3 inches = **4800 steps**. Since the speed setting is 1:1, the cursor moves 4800 pixels; this is more than twice the width of a screen that is 1920 pixels wide.
2. **Polling interval.** 500 transmissions per second: 1000 ms ÷ 500 = once every **2 ms**. For a 1000 Hz mouse this falls to 1 ms.

A high DPI is not always better. At a very high DPI even the slightest hand tremor moves the cursor several pixels; precise work becomes harder. A suitable value is chosen according to the screen's resolution and the user's habits. The mouse's buttons and wheel also do jobs such as selecting, opening the context menu, and scrolling the page.

There are also pointing devices that are alternatives to the mouse: the **touchpad** on laptops, the **stylus** for drawing, the **trackball** that works with a ball turned by hand, and the **touch screen**, which is touched directly. The touch screen, as will be seen in the next section, is also an output device.

1.2.8 Output devices

Output devices turn the digital result a computer produces into a form people can perceive: light, sound, ink on paper, or vibration.

1.2.8.1 Output devices and their classification

Definition 1.2.19

An **output device** is a device that presents the data a computer produces to the user as an image, sound, print, or movement.

Device	Output produced
Monitor	text, images, and video
Printer	a copy of the digital document on paper
Speaker, headphones	sound
Projector	an image projected onto a large surface
Haptic feedback	vibration or force

Device	Output produced
Multifunction device	input and output together (e.g., a printer with a scanner)

Table 1.2.11: Main output devices.

The only criterion that decides whether a device is input or output is **the direction of the data flow**. If data flows toward the computer, the device is an input device; if it flows toward the user, it is an output device. In some devices data flows both ways (Figure 1.2.22).

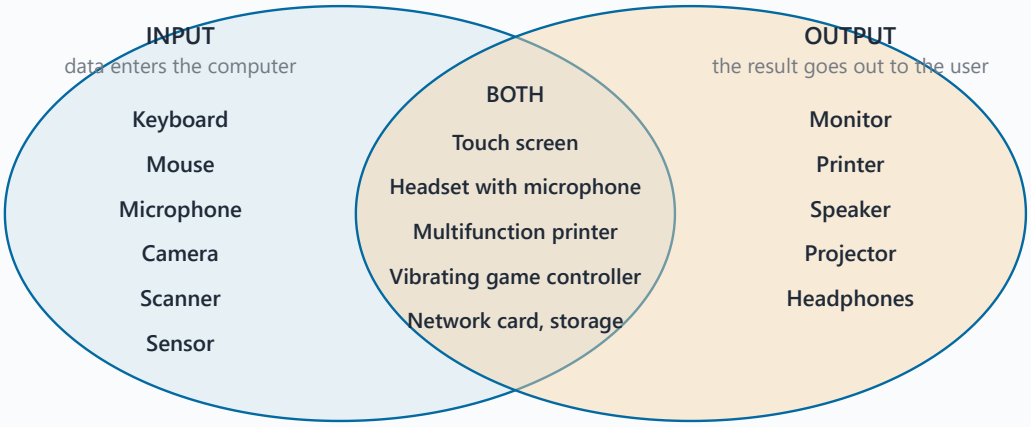


Figure 1.2.22: Input devices, output devices, and devices that are both. The touch screen, headset with microphone, multifunction printer, vibrating game controller, network card, and storage devices work in both directions.

Example 1.2.15

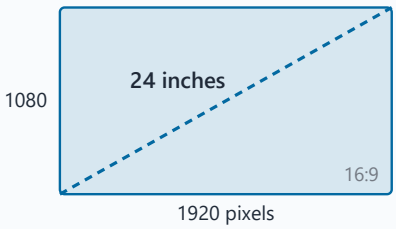
Which group do a webcam, a smartwatch screen (touch), a printer, an external SSD, and a game steering wheel (with force feedback) belong to?

- **Webcam:** gives images to the computer → input.
- **Smartwatch screen:** takes touch, shows images → both.
- **Printer:** puts the document on paper → output (for a printer that only prints).
- **External SSD:** both read and written → both.
- **Game steering wheel:** sends the turning angle and returns the bumps of the road to the wheel as force → both.

1.2.8.2 The monitor: resolution, size, and aspect ratio

A monitor forms an image by dividing the screen surface into millions of small dots. Each of these dots is called a **pixel**. Four basic values appear in a monitor's specifications:

- **Resolution:** the number of horizontal and vertical pixels; for example 1920 × 1080.
- **Screen size:** the length of the diagonal, measured in inches; for example 24 inches.
- **Pixel density:** how many pixels fall on one inch (1.2.8.3).
- **Aspect ratio:** the ratio of the screen's width to its height; for example 16:9.



Pixels along the diagonal:

$\sqrt{(1920^2 + 1080^2)} \approx 2203$ pixels

Pixel density:

$2203 \div 24 \approx 92$ PPI

The same 1920 × 1080 on a 15.6-inch laptop ≈ 141 PPI

Figure 1.2.23: A 24-inch, 16:9 monitor: 1920 pixels wide and 1080 pixels high. The number of pixels along the diagonal is $\sqrt{(1920^2 + 1080^2)} \approx 2203$; the pixel density is $2203 \div 24 \approx 92$ PPI. The same resolution on a 15.6-inch laptop gives about 141 PPI.

The aspect ratio of a 1920 × 1080 resolution is 16:9, because $1920 \div 1080 = 16 \div 9 \approx 1.78$. Because the screen size is given as a diagonal, the width and height are not written directly; but they can be calculated from the aspect ratio.

Example 1.2.16

How many cm wide and high is the screen of a 24-inch, 16:9 monitor?

1. **The diagonal of the ratio.** If the width is taken as 16 units and the height as 9 units, by the Pythagorean theorem the diagonal is $\sqrt{(16^2 + 9^2)} = \sqrt{(256 + 81)} = \sqrt{337} \approx 18.36$ units.
2. **How many inches is one unit?** Since the diagonal is 24 inches, 1 unit = $24 \div 18.36 \approx 1.307$ inches.
3. **Width and height.** The width is $16 \times 1.307 \approx 20.92$ inches and the height is $9 \times 1.307 \approx 11.77$ inches.
4. **Converting to centimeters.** 1 inch = 2.54 cm: width ≈ $20.92 \times 2.54 \approx$ **53.1 cm**, height ≈ $11.77 \times 2.54 \approx$ **29.9 cm**.

1.2.8.3 Pixel density

The same resolution looks sharper on a small screen. The reason is that the same number of pixels is fitted into a smaller area. As pixels get smaller, the eye cannot tell them apart one by one, and edges look smoother (Figure 1.2.24).

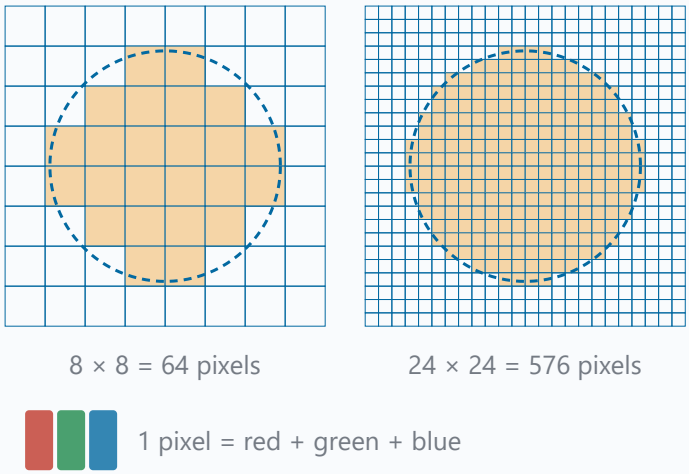


Figure 1.2.24: The same circle on grids of 8 × 8 = 64 and 24 × 24 = 576 pixels. As the number of pixels increases, the edges look smoother. Each pixel consists of three subpixels: red, green, and blue.

The three-color bar under the figure shows an important detail: a pixel actually consists of three small light sources. Every color is obtained by adjusting the brightness of the red, green, and blue subpixels. If all three are fully bright, it looks white; if all three are dim, black.

Definition 1.2.20

Pixel density is the number of pixels per inch of the screen's length and is measured in **PPI** (pixels per inch). It is calculated along the diagonal:

$$\text{PPI} = \sqrt{(\text{horizontal pixels}^2 + \text{vertical pixels}^2)} \div \text{diagonal (inches)}$$

The logic of the formula is this: the length of the diagonal is known in inches; how many pixels lie along the diagonal is found with the Pythagorean theorem. Dividing the second by the first gives the number of pixels per inch.

Example 1.2.17

What is the PPI of a 27-inch, 3840 × 2160 (4K) monitor? How much denser is it than a 2560 × 1440 monitor of the same size?

1. **Diagonal pixels of 4K.** $3840^2 = 14{,}745{,}600$ and $2160^2 = 4{,}665{,}600$. The total is 19,411,200; its square root ≈ 4405.8.
2. **4K PPI.** 4405.8 ÷ 27 ≈ **163.2 PPI**.
3. **Diagonal pixels of 2560 × 1440.** $2560^2 + 1440^2 = 6{,}553{,}600 + 2{,}073{,}600 = 8{,}627{,}200$; its square root ≈ 2937.2.

4. **2560 × 1440 PPI.** 2937.2 ÷ 27 ≈ **108.8 PPI**.

At the same size the 4K screen is about 1.5 times denser. The difference in the number of pixels is larger: 3840 × 2160 = 8,294,400 pixels is 2.25 times 2560 × 1440 = 3,686,400 pixels. So a 4K screen also taxes the graphics card more.

Common mistake

Treating resolution alone as a measure of sharpness. A 32-inch 4K monitor is about 138 PPI; a 14-inch 1920 × 1080 laptop is about 157 PPI. The laptop screen with the lower resolution looks sharper. What determines sharpness is the pixel density that resolution and size give together; viewing distance also plays a part.

1.2.8.4 Refresh rate and response time

Motion on the screen actually consists of still frames shown one after another. If the frames change fast enough, the eye perceives them as continuous motion. Several properties determine how well a monitor shows motion:

Property	What it means
Refresh rate	How many times per second the screen is refreshed; measured in Hz.
Response time	The time a pixel takes to change from one color to another.
Brightness	The screen's light output (cd/m^2).
Color accuracy	How close the colors shown are to the target values.
Viewing angle	How well the image holds up when viewed from the side.

Table 1.2.12: Motion and image properties of a monitor.

A screen with a **refresh rate** of 60 Hz refreshes its image 60 times per second. The time a frame stays on the screen is the inverse of the refresh rate: 1000 ms ÷ 60 ≈ 16.7 ms. As the refresh rate rises, this time shortens and motion looks smoother (Figure 1.2.25).

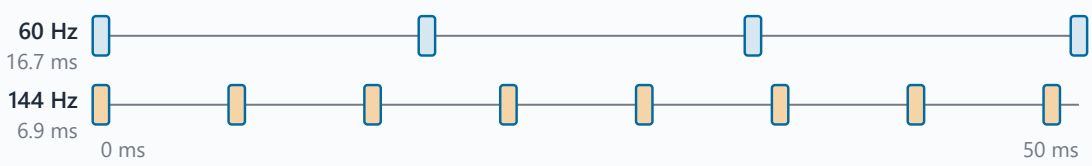


Figure 1.2.25: In 50 ms a 60 Hz screen shows 3 new frames and a 144 Hz screen shows 7. The duration of a frame is 16.7 ms at 60 Hz and 6.9 ms at 144 Hz.

Example 1.2.18

On 60, 144, and 240 Hz screens, how many ms does a frame stay on the screen? How many new frames does each show in 50 ms?

Refresh rate	Frame duration (1000 ÷ Hz)	Frames in 50 ms (50 ÷ duration)
60 Hz	≈ 16.7 ms	3
144 Hz	≈ 6.9 ms	7.2 → 7 full frames
240 Hz	≈ 4.2 ms	12

Even if a screen's refresh rate is high, there is no gain if the graphics card cannot produce that many frames. For a 144 Hz monitor to be useful, the game or application must be able to produce close to 144 frames per second.

Response time measures something else: how long a pixel takes to reach a new color after it is told to. On a screen with a long response time, a faint trail remains behind fast-moving objects. The two properties complement each other: the refresh rate determines how often new frames arrive, and the response time determines how cleanly each frame is drawn.

Common mistake

Taking refresh rate and response time to be the same thing. On a 144 Hz screen the frame duration is 6.9 ms; but the same screen's response time may be 1 ms or 5 ms. One measures the number of frames per second, the other the speed at which a pixel changes color.

1.2.8.5 Panel technologies: LCD and OLED

How a monitor produces its light determines the image's contrast, colors, and viewing angle. Two main technologies are used today: LCD and OLED.

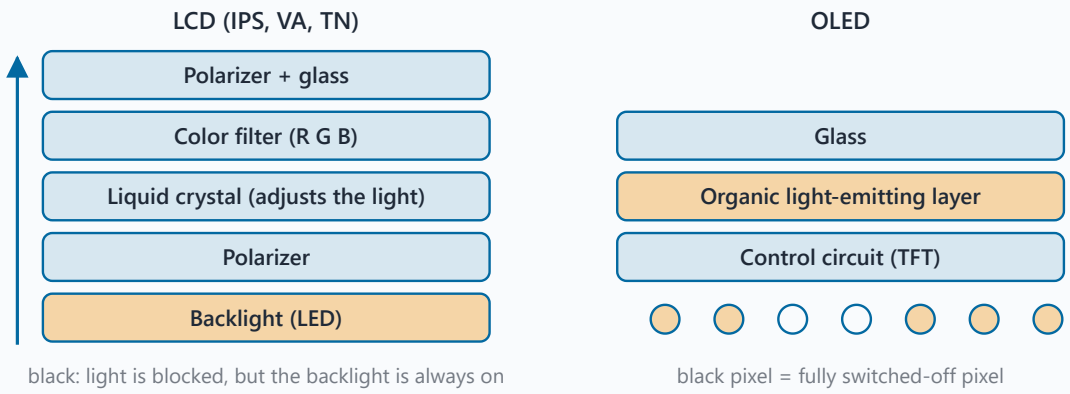


Figure 1.2.26: LCD and OLED layers. In an LCD there is an always-on backlight (LED) at the bottom; light passes through the polarizer, the liquid crystal, and the color filter. For black, the liquid crystal tries to block the light, but the backlight stays on. In an OLED each pixel produces its own light; a black pixel is fully off.

In an **LCD** (liquid-crystal display) panel, a **backlight** produces the light. The **liquid crystal** layer in front of it works like a small shutter that adjusts the amount of light passing through for each subpixel. A color filter gives the passing light its red, green, or blue color. To show black, the shutter closes; but the shutter cannot block the light completely and some leaks through. This is why an LCD's black looks dark gray in a dark room.

An **OLED** (organic light-emitting diode) panel has no backlight. Each pixel produces its own light. A black pixel is completely off; it emits no light at all. So an OLED's contrast is very high and black is truly black.

LCD panels themselves come in several types. IPS, VA, and TN are types of LCD; the difference between them lies in how the liquid crystals are arranged:

Panel	Strengths	Weaknesses
IPS (LCD)	wide viewing angle, balanced colors	medium black level
VA (LCD)	high contrast, deep blacks	narrower viewing angle than IPS
TN (LCD)	low response time	limited viewing angle and colors
OLED	each pixel produces its own light, very high contrast	usually more expensive

Table 1.2.13: Panel technologies.

The panel is chosen according to the purpose of use. For office work and design, color accuracy and viewing angle matter; this is why IPS is common. For watching films in a dark room, contrast stands out; VA and OLED are strong here. In competitive games, response time and refresh rate are decisive. The budget is a separate limit on every choice.

1.2.8.6 Printers, sound, and projectors

Printers transfer a digital document to paper. Two technologies are common today (Figure 1.2.27):

- An **inkjet printer** sprays microscopic ink droplets onto the paper through tiny nozzles in its head. It is common for color documents and photo printing.
- A **laser printer** uses a laser beam to form a pattern of electric charge on a rotating drum. Powdered pigment (toner) sticks to this pattern, is transferred to the paper, and is fixed to the paper with heat. It can be fast and economical for high-volume text printing.

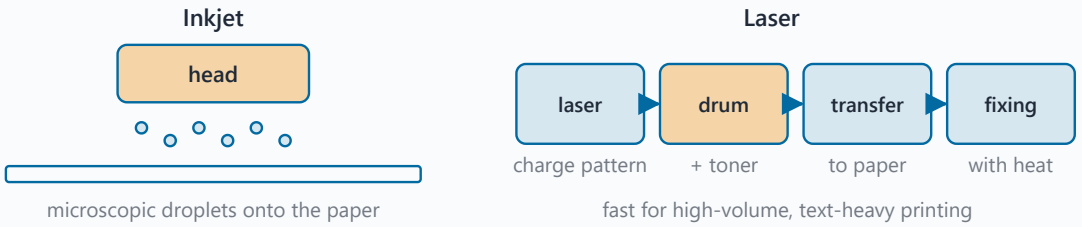


Figure 1.2.27: An inkjet printer sprays droplets onto the paper. In a laser printer a laser forms a charge pattern on the drum; toner sticks to this pattern, is transferred to the paper, and is fixed with heat.

Speakers and headphones turn the analog signal from the sound card's DAC, or data from a digital connection such as USB or Bluetooth, into sound. With a digital connection, the DAC is inside the speaker or the headphones.

A **projector** casts the image onto a screen or a wall. Three values stand out when choosing one: brightness, resolution, and throw distance. Brightness must be high for the image to be visible in a well-lit classroom. Throw distance shows how far from the screen the projector must be placed for a given image size.

1.2.9 Storage

Everything in RAM is lost when the computer is switched off. A separate unit is needed to keep documents, photographs, programs, and the operating system permanently. This section covers storage devices, the speed differences between them, and what those differences mean in everyday use.

Definition 1.2.21

A **storage** device is **non-volatile** memory, that is, memory that keeps data even when the power is cut. Its main types are the hard disk (HDD), the solid-state drive (SSD), the USB flash drive, the memory card, and the optical disc.

1.2.9.1 A document's journey between components

The best way to see storage's place in the system is to follow an everyday task. A document called `report.docx` is opened, edited, and saved. During this, data travels among the mouse, the SSD, RAM, the processor, and the screen (Figure 1.2.28).

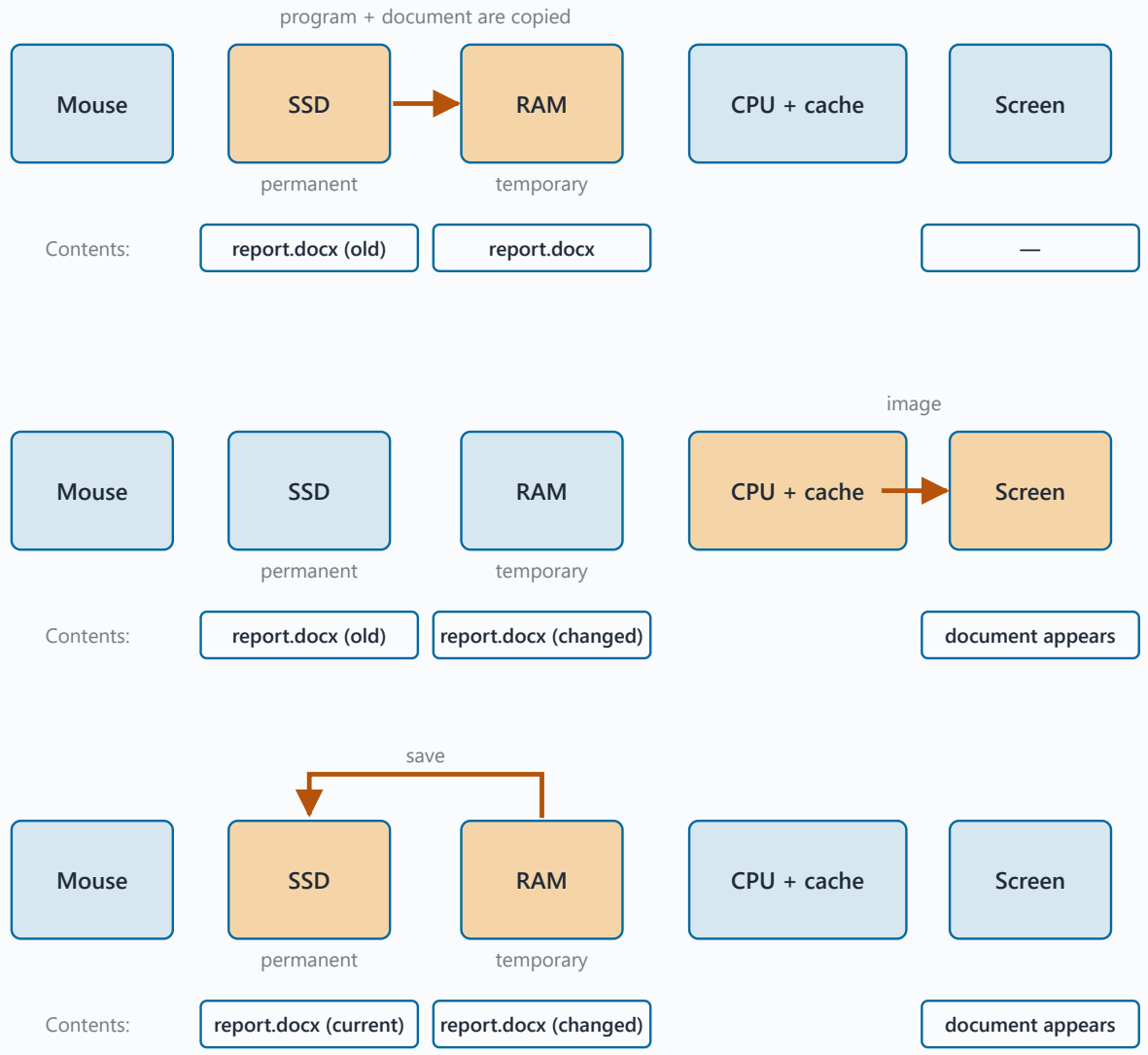


Figure 1.2.28: Opening and saving a document, three moments. At the top, the program and the document are copied from the SSD to RAM. In the middle, the document has been edited: the copy in RAM has changed, the copy on the SSD is still old; the result appears on the screen. At the bottom, "Save" writes the changes from RAM to the SSD and the copy on the SSD is updated.

Step	What happens?	Document on the SSD	Document in RAM
1	The user double-clicks `report.docx`; the mouse passes this event to the computer (input).	old	—
2	The operating system finds the program and document files on the SSD.	old	—
3	The program and the document are copied from the SSD to RAM.	old	old (copy)
4	The CPU brings instructions from RAM into the cache and registers and processes them (fetch–decode–execute).	old	being edited
5	The result goes through the graphics card to the monitor; the user sees the document and edits it (output).	old	changed
6	With "Save," the changes are written from RAM to the SSD.	current	changed

Table 1.2.14: Steps of opening, editing, and saving a document.

Step 3 of the table shows an important fact: **the CPU does not run the program directly from the SSD**. The program is first copied to RAM; the processor fetches instructions from RAM through the cache. The SSD is too slow for the processor to put up with the wait (1.2.4.2). The read path therefore follows the order SSD → RAM → cache → registers; saving goes in the direction RAM → SSD.

In steps 4 and 5 there are two different versions of the document: the edited copy in RAM and the old copy on the SSD. If the power goes off at this point, the changes in RAM are lost and the old version on the SSD remains. This is why many programs save automatically at regular intervals.

Common mistake

Thinking that the document seen on the screen sits on the disk in the same state. The image on the screen reflects the copy in RAM. Until it is saved, the file on the disk is in its old state.

1.2.9.2 HDD and SSD

The two main storage technologies in computers store data in completely different ways.

Definition 1.2.22

A **hard disk drive** (HDD) is a storage device that keeps data on rotating magnetic **platters**. Data is read and written by a **read–write head** at the end of an arm. An **SSD** (solid-state drive) stores data in **flash memory** chips that have no moving parts.

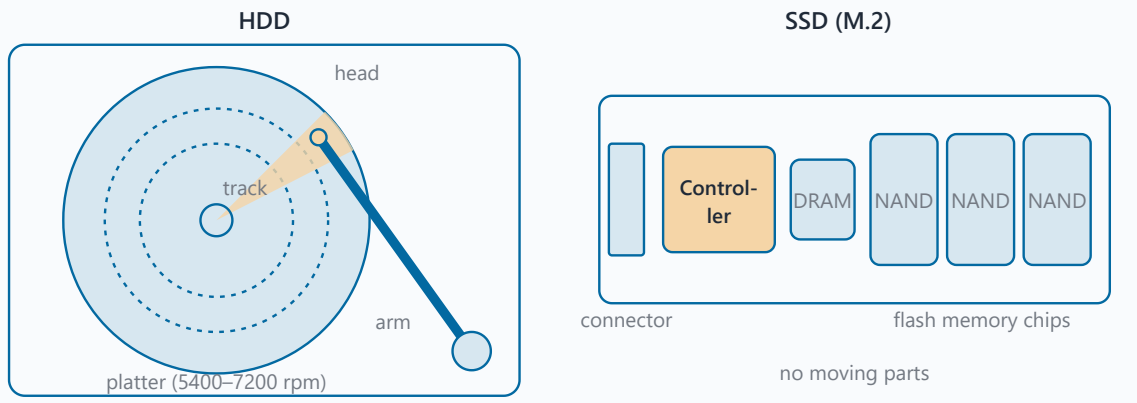


Figure 1.2.29: On the left, the inside of an HDD: the platter spinning at 5400–7200 rpm, the tracks on the platter, the arm, and the head. On the right, an M.2 SSD: the controller, the DRAM used as a cache, and the NAND flash memory chips. An SSD has no moving parts.

Reading data on an HDD is a mechanical job. The head must move to the right **track** (one of the concentric rings on the platter), and then the **sector** to be read (a small piece of the track) must rotate around under the head. These two movements take milliseconds. In an SSD, the **controller** reads data from several flash chips at the same time; access is on the order of microseconds.

Property	HDD	SSD
Technology	rotating magnetic disk	flash memory
Speed	lower	higher
Shock resistance	more sensitive because of the mechanical parts	no moving parts
Cost per capacity	usually lower	usually higher
Common use	archives and high capacity	operating system and applications

Table 1.2.15: Comparison of HDD and SSD.

A common arrangement follows from this comparison. The operating system and programs are put on an SSD so that they open quickly. Large but rarely used data such as photo archives and backups are kept on the cheaper HDD. In laptops, almost only SSDs are used because of their shock resistance, low power consumption, and silence.

1.2.9.3 SATA and NVMe

The speed of an SSD does not depend only on the flash chips; the path by which the SSD is connected to the computer is also decisive. Two interfaces are common.

Property	SATA SSD	NVMe SSD
Interface	SATA (an older interface designed for hard disks)	the NVMe protocol over PCIe

Property	SATA SSD	NVMe SSD
Typical sequential read	about 550 MB/s (the SATA III limit)	a few GB/s (depending on the PCIe version and the number of lanes)
Parallelism	a single command queue	many parallel queues; low latency
Form	2.5-inch case or M.2	mostly M.2 (U.2 in some servers)

Table 1.2.16: SATA SSD and NVMe SSD.

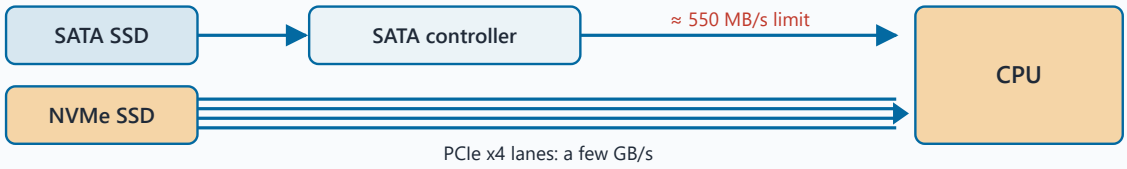


Figure 1.2.30: The data of a SATA SSD passes through the SATA controller and is limited to about 550 MB/s. An NVMe SSD, on the other hand, is connected directly to the processor with four PCIe lanes and can transfer a few GB per second.

SATA was designed for the speed of spinning disks. SSDs soon exceeded this limit: the theoretical limit of SATA III is 6 Gbit/s; when the bits reserved for data encoding are subtracted, this comes to about 600 MB/s. This is why SATA SSDs stay around 550 MB/s in practice. NVMe (Non-Volatile Memory Express) is a protocol designed directly for flash memory and uses PCIe lanes.

Example 1.2.19

A PCIe lane carries about 0.985 GB per second in PCIe 3.0, and this value doubles with each new generation. What is the theoretical limit of a four-lane (x4) NVMe SSD in each generation?

1. **PCIe 3.0 x4.** $4 \times 0.985 \approx$ **3.94 GB/s**.
2. **PCIe 4.0 x4.** Per lane $2 \times 0.985 \approx 1.97$ GB/s; $4 \times 1.97 \approx$ **7.88 GB/s**.
3. **PCIe 5.0 x4.** Per lane $2 \times 1.97 \approx 3.94$ GB/s; $4 \times 3.94 \approx$ **15.75 GB/s**.

Compared with SATA's limit of 0.6 GB/s, a PCIe 4.0 x4 connection is about 13 times wider. Real SSDs can approach these limits but cannot exceed them.

Common mistake

Thinking M.2 is a protocol and that every M.2 SSD will run at NVMe speed. M.2 is only a form factor, that is, the shape of the card and its slot. An M.2 slot may support SATA, NVMe, or both; which is supported is written in the motherboard manual. A SATA SSD in the M.2 form stays at the 550 MB/s limit.

1.2.9.4 Reading a sector on an HDD

To see why an HDD is slow, let us follow the reading of a single sector step by step. The platter spins at 7200 revolutions per minute. The sector to be read and the read–write head are initially in different places. The access time consists of three parts (Figure 1.2.31).

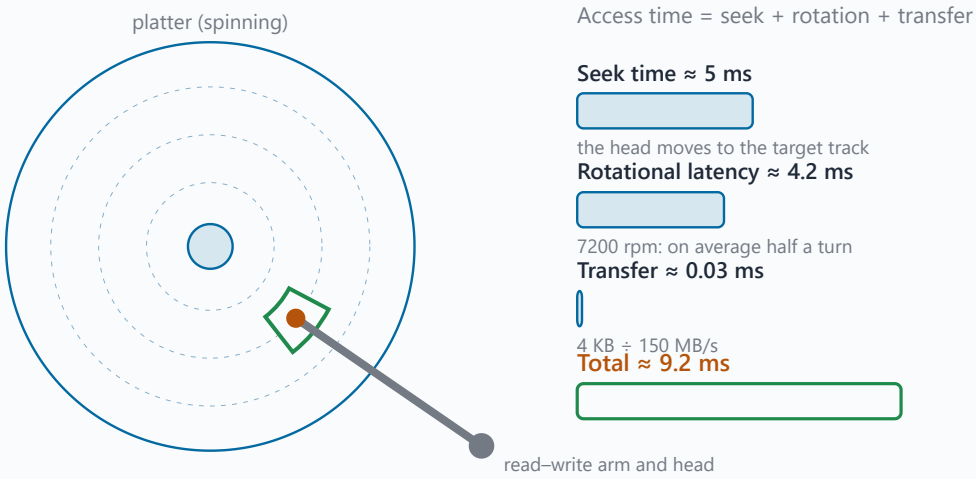


Figure 1.2.31: Reading a sector on an HDD. The head first moves to the target track (seek, ≈ 5 ms); then it waits for the sector to rotate under the head (rotational latency, ≈ 4.2 ms); finally the data is read (transfer, ≈ 0.03 ms). Total ≈ 9.2 ms.

Definition 1.2.23

The **access time** of a sector on an HDD is the sum of three parts:

> access time = seek time + rotational latency + transfer time

Seek time is the time the arm takes to bring the head to the target track. **Rotational latency** is the time expected for the sector to rotate under the head; on average, half a turn is assumed. **Transfer time** is the time to read the data while the sector passes under the head.

Example 1.2.20

On an HDD spinning at 7200 rpm, the average seek time is 5 ms and the transfer rate is 150 MB/s. What is the average access time of a 4 KB sector? (Take 1 MB = 1000 KB.)

1. **Seek.** The arm brings the head to the target track: **5 ms**.
2. **One full turn.** 1 minute = 60,000 ms. 60,000 ÷ 7200 ≈ 8.33 ms per turn.
3. **Rotational latency.** On average the sector is half a turn away: 8.33 ÷ 2 ≈ **4.17 ms** (about 4.2 ms).

4. **Transfer.** 4 KB = 0.004 MB. 0.004 ÷ 150 ≈ 0.0000267 s ≈ **0.03 ms**.
5. **Total.** 5 + 4.17 + 0.03 ≈ **9.2 ms**.

Of the total, 9.17 ms is mechanical waiting; the actual reading is 0.03 ms. The waiting is more than 300 times the reading.

This calculation explains why an HDD is acceptable for large, single-piece files but very slow for many small files. A large file is stored in sectors lying one after another on the platter: the head positions itself once and the data is read continuously. Each of 100 small files, however, is in a separate place. The wait for seek and rotation is lived again for each file: 100 × 9.2 ms = 920 ms, almost a second. The same job takes 100 × 0.1 = 10 ms on an SSD with an access time of about 0.1 ms.

Common mistake

Taking the full-turn time as the rotational latency. The sector may be right in front of the head or a full turn away. On average, half a turn is waited; this is why the turn time is divided by 2.

1.2.9.5 Copy time and capacity units

The time to copy a large file can be estimated from the storage device's **sequential read** speed. Sequential reading is reading data continuously from blocks that lie one after another; for large files this speed is decisive.

> time = file size ÷ transfer rate

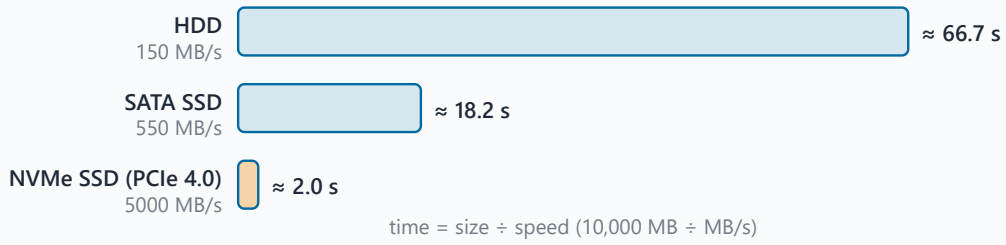


Figure 1.2.32: Copy time of a 10 GB (10,000 MB) file: HDD (150 MB/s) ≈ 66.7 s, SATA SSD (550 MB/s) ≈ 18.2 s, NVMe SSD (PCIe 4.0, 5000 MB/s) ≈ 2.0 s.

Example 1.2.21

A 10 GB video file is read from three different devices. What are the times?

1. **Unit conversion.** With the base manufacturers use, 10 GB = 10,000 MB.

2. **HDD.** 10,000 ÷ 150 ≈ **66.7 s**.
3. **SATA SSD.** 10,000 ÷ 550 ≈ **18.2 s**.
4. **NVMe SSD.** 10,000 ÷ 5000 = **2.0 s**.

These values hold for large, single-piece files. With many small files an HDD slows down much more because of head movement (1.2.9.4). Copying needs both reading and writing: the time depends on the slower of the source and target devices and on the connection between them (USB, network); data flows at the speed of the slowest link in this chain.

There are two separate conventions for capacity units, and this difference often causes confusion. Disk manufacturers use the decimal base with SI prefixes: 1 GB = 1000 MB = 10^9 bytes. Operating systems often compute capacity with a base of 1024: 1 GiB (gibibyte) = 1024 MiB = 2^{30} = 1,073,741,824 bytes. The trouble is that some operating systems display this second value as "GB" instead of "GiB."

Example 1.2.22

How many GB does a disk labeled "1 TB" on the box show in the operating system?

1. **The manufacturer's value.** 1 TB = 10^{12} = 1,000,000,000,000 bytes.
2. **Converting to base 1024.** 1 GiB = 2^{30} = 1,073,741,824 bytes. 10^{12} ÷ 1,073,741,824 ≈ **931.3 GiB**.

If the operating system displays this value as "931 GB," the disk looks about 7% "smaller." The byte count is the same; only the unit changes. By the same calculation, a 500 GB disk shows as ≈ 465.7 GiB and a 256 GB SSD as ≈ 238.4 GiB.

1.2.9.6 Portable and archival storage

Besides the disk inside the computer, there are other storage options for carrying, backing up, or sharing data.

Option	Use and characteristics
USB flash drive	Easy to carry; there is a risk of loss and physical damage.
SD card	Common in cameras and mobile devices.
Optical disc	Used for software distribution and in certain archival scenarios.
External disk	Offers high capacity for backup and moving data.
Network storage	Gives several users on the local network central access to files.
Cloud storage	Provides synchronization and sharing over the internet.

Table 1.2.17: Portable and archival storage options.

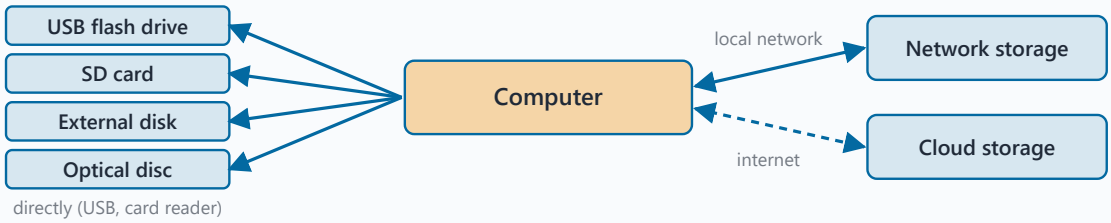


Figure 1.2.33: The computer connects directly (by USB or a card reader) to USB flash drives, SD cards, external disks, and optical discs; it reaches network storage over the local network and cloud storage over the internet.

The most important use of these options is **backup**. If a file has only one copy, a failing disk, a stolen computer, or an accidental deletion destroys that file permanently. A common recommendation is to keep at least three copies of important data, on at least two different media, and to keep one of these copies in another place (for example in the cloud).

1.2.10 System design

Choosing good parts one by one is not enough to get a good computer. The parts must fit one another, must not limit one another, and must serve the purpose of use. This section collects the criteria for evaluating a system as a whole.

1.2.10.1 Component compatibility

Definition 1.2.24

Compatibility is the ability of components to work together without trouble in physical, electrical, logical, and software terms.

Kind	What is checked
Physical	The sizes of the case, motherboard, and cards must fit one another.
Electrical	The power supply's capacity and connections must be sufficient.
Logical	The processor socket, the memory generation, and the chipset must be supported.
Connection	Port, cable, and protocol versions must be compatible.
Thermal	The cooling system must handle the heat load of the processor and the GPU.
Software	Operating system and driver support must be available.

Table 1.2.18: Kinds of compatibility.

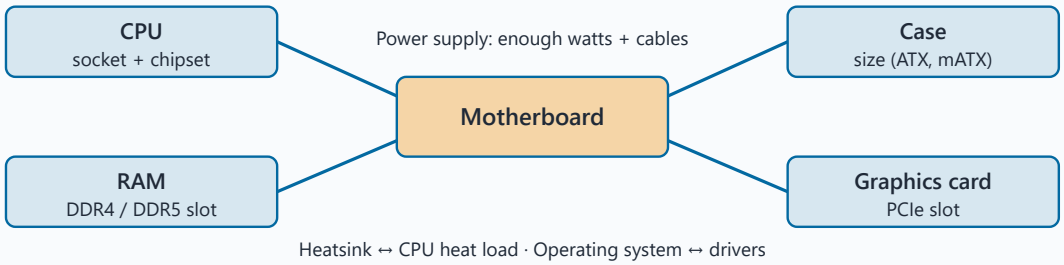


Figure 1.2.34: Compatibility checks gather around the motherboard: CPU with the socket and chipset, RAM with the DDR4/DDR5 slot, the case with the motherboard size (ATX, mATX), the graphics card with the PCIe slot. In addition, the power supply must provide enough watts and cables, the heatsink must handle the processor's heat load, and drivers must exist for the operating system.

The motherboard is at the center of the compatibility check (Figure 1.2.34). When a computer is assembled part by part, the checks are made roughly in a certain order. First the processor and a motherboard with a matching socket are chosen. Then the RAM generation the motherboard supports is determined. Next one checks whether the motherboard and the graphics card fit in the case. Last, the power and cooling needs of the whole system are calculated.

Example 1.2.23

A student has chosen an ATX-size motherboard that supports DDR5. The motherboard's M.2 slot supports only SATA. The student's other chosen parts are two DDR4 RAM modules, an NVMe SSD, and an mATX-size case. What problems are there?

1. **RAM.** DDR4 and DDR5 modules are physically different; a DDR4 module does not fit a DDR5 slot. The RAM must be changed to DDR5.
2. **SSD.** If the M.2 slot supports only SATA, an NVMe SSD will not work in this slot. Either a SATA M.2 SSD must be chosen or a slot that supports NVMe must be used.
3. **Case.** An mATX case does not take the larger ATX motherboard. The case must be ATX; the reverse (an mATX motherboard in an ATX case) is fine.

None of the three problems is about the quality of the individual parts; all of them are compatibility problems.

1.2.10.2 DDR5 and current standards

Computer hardware is updated with new standards every few years. Which standard a part supports determines both its speed and its compatibility with the other parts.

Standard	What changed?	What to watch for
DDR4 → DDR5	Bandwidth and the speed range increase; DDR5 modules start at higher speeds. Power management is on the module itself; each module has two independent subchannels.	Physically different: a DDR4 module does not fit a DDR5 slot (and vice versa).
PCIe 3 → 4 → 5	Bandwidth per lane roughly doubles with each generation.	The slot is backward compatible; the speed is set by the slower of the slot and the device.
USB 3.2 → USB4	USB-C connector, 20/40 Gbit/s, and Thunderbolt compatibility; USB4 Version 2.0 defines even higher speeds.	Not every USB-C port supports these speeds.
Wi-Fi 6 → 7	6 GHz band (introduced with 6E), channel width up to 320 MHz, multi-link operation (MLO).	Both the router and the device must support it.

Table 1.2.19: Changes in current standards.

Some of the terms in the table should be explained. **Bandwidth** is the amount of data a connection can carry per unit of time. **Backward compatible** means that a new standard can also work with old devices. The **channel width** in Wi-Fi is the range of frequencies the wireless transmission uses; a wide channel means more data carried at once. MLO (Multi-Link Operation) is a device's using several frequency bands at the same time.

Example 1.2.24

A PCIe 4.0 x4 NVMe SSD is plugged into a PCIe 3.0 x4 M.2 slot. Does the SSD work? What is the theoretical top speed?

1. **Operation.** PCIe is backward compatible; the SSD works.
2. **Speed.** The speed is set by the slower of the slot and the device: PCIe 3.0. By the calculation in Example 1.2.19, the 3.0 x4 limit is ≈ 3.94 GB/s. Even if the SSD's own limit is 7 GB/s, in this slot it reaches at most about 3.9 GB/s.

Common mistake

Thinking that DDR generations are backward compatible like PCIe. PCIe cards work in the slots of different generations, but the notches of DDR4 and DDR5 modules are in different places, and they cannot be physically plugged into each other's slots.

1.2.10.3 Choosing by purpose

There is no "best" computer for every user; there is the computer best suited to the purpose of use. When the budget is limited, money is directed to the components the job needs most.

Use	Priority components
Office and education	balanced CPU, enough RAM, SSD, ergonomic screen
Gaming	GPU, CPU, cooling, high-refresh-rate screen
Content creation	multicore CPU, GPU, RAM, color accuracy, fast storage
Data analysis	RAM capacity, CPU/GPU, fast storage
Portable use	battery, weight, screen, connections, energy efficiency

Table 1.2.20: Priority components by use.

The logic of the table rests on what was explained in the earlier sections. Office work is lightweight and single-threaded; many cores or a powerful GPU make little difference, but an SSD makes programs open quickly. In games every frame is drawn on the GPU, so the graphics card is decisive; a high-refresh-rate screen is useful only if the GPU can produce enough frames. Content-creation jobs such as video editing can be split up; they benefit directly from many cores and fast storage. Programs that process large datasets want to keep the data in RAM; if RAM is not enough, the system uses the disk and slows down noticeably.

Example 1.2.25

There are two options for the same budget. (A) A powerful discrete GPU, 8 GB of RAM, and a 1 TB HDD. (B) An integrated GPU, 16 GB of RAM, and a 512 GB NVMe SSD. Which suits a student who takes lecture notes, does research on the internet, and keeps several programs open at the same time?

1. **Workload.** Text editing, the browser, and office programs put little load on the GPU; an integrated GPU is enough.
2. **RAM.** Having many browser tabs and programs open at once needs RAM. With 8 GB the system may start using swap space; 16 GB is more comfortable.
3. **Storage.** Startup of the operating system and programs depends on storage speed. An NVMe SSD is much faster than an HDD (1.2.9.5).

B is more suitable for this use. Option A is strong for gaming, but in this student's daily tasks it would feel slower because of the slow disk and little RAM.

1.2.11 Summary

- **Hardware** is the physical parts of a computer; **software** tells these parts what to do. Every task consists of the steps input, processing, memory, storage, and output.
- The **CPU** processes instructions in the **fetch–decode–execute** cycle. The ALU calculates, the control unit manages the flow, and the registers hold the current values. The duration of a clock cycle is 1 ÷ clock speed (≈ 0.33 ns at 3 GHz). Performance is determined by clock speed, number of

cores, cache, and architecture together; “8 cores = 8 times the speed” is not true.

- **x86-64** is common in desktops and servers and **ARM** in mobile devices; **RISC-V** is an open instruction set. A **SoC** gathers the CPU, GPU, NPU, and controllers on a single chip.
- The **motherboard** connects the components and determines compatibility (socket, DIMM, PCIe, M.2, SATA, chipset). **RAM** is temporary and fast; every byte has an address. The **UEFI** in **ROM/flash** initializes the hardware at startup and loads the operating system.
- In the **memory hierarchy**, going up, speed increases and capacity decreases: register < 1 ns, cache 1–10 ns, RAM 50–100 ns, SSD ≈ 0.1 ms, HDD 5–10 ms. An SSD is 1000–2000 times slower than RAM; an HDD is 50–100 times slower than an SSD.
- In the **cache**, finding the data is a **hit** (≈ 1 ns) and not finding it is a **miss** (≈ 80 ns); on a miss the whole cache line is fetched. Sequential access exploits **spatial locality**, and repeated access to the same data exploits **temporal locality**. Summing an 8-element array takes 166 ns with a cache and 640 ns without.
- A **GPU** works in parallel with thousands of simple cores; an integrated GPU shares RAM, while a discrete GPU has its own VRAM. HDMI and DisplayPort carry digital video, VGA analog. In USB, the connector shape and the version (speed) are separate; a speed in bits is divided by 8 to convert to bytes.
- In the **sound card**, the ADC samples analog sound into numbers and the DAC turns numbers back into a signal. Recording size = sampling rate × sample size × channels × duration.
- The **case** provides airflow; an overheating processor slows down by **thermal throttling**. The **power supply** converts AC to DC; its capacity is chosen above the total need with a safe margin (420 W × 1.3 = 546 W → 550 W).
- Whether a device is **input** or **output** is determined by the direction of the data flow; the touch screen, storage, and network card work in both directions.
- **PPI** = $\sqrt{(\text{horizontal}^2 + \text{vertical}^2)} \div \text{diagonal}$; resolution and size together determine sharpness. Frame duration = 1000 ÷ Hz in ms; refresh rate and response time measure different things. An LCD uses a backlight; in an OLED each pixel produces its own light.
- The CPU runs the program from RAM, not from the SSD; unsaved changes are in RAM. **HDD** access = seek + rotational latency (half a turn) + transfer, and takes milliseconds. **NVMe** exceeds SATA’s ≈ 550 MB/s limit over PCIe; M.2 is a form factor, not a protocol. 1 TB = 10^{12} bytes ≈ 931.3 GiB.
- In system design, **compatibility** is checked in physical, electrical, logical, connection, thermal, and software terms. PCIe is backward compatible, DDR4 and DDR5 are not. Components are prioritized by purpose of use.

Calculation	Formula	Example
Clock cycle	1 ÷ clock speed	3.2 GHz → 0.3125 ns
Time with a cache	misses × 80 + hits × 1 (ns)	2 misses, 6 hits → 166 ns
Bits → bytes	Mbit/s ÷ 8 = MB/s	480 Mbit/s → 60 MB/s
Sound recording	samples/s × bytes × channels × s	48,000 × 2 × 1 × 30 = 2,880,000 bytes

Calculation	Formula	Example
Power supply	total × 1.3, model above	420 W → 546 W → 550 W
Pixel density	$\sqrt{(w^2 + h^2)} \div$ diagonal	1920 × 1080, 24 inches → ≈ 91.8 PPI
Frame duration	1000 ÷ Hz (ms)	144 Hz → ≈ 6.9 ms
HDD access	seek + half a turn + transfer	5 + 4.17 + 0.03 ≈ 9.2 ms
Copying	size ÷ speed	10,000 MB ÷ 550 MB/s ≈ 18.2 s
GB → GiB	bytes $\div 2^{30}$	1 TB → ≈ 931.3 GiB

Table 1.2.21: Main calculations used in these notes.

1.2.12 Key terms

hardware central processing unit (CPU) core clock speed clock cycle thread instruction set System-on-Chip (SoC) neural processing unit (NPU) motherboard chipset slot lane (PCIe) random access memory (RAM) volatile / non-volatile swap space read-only memory (ROM) firmware BIOS / UEFI power-on self-test (POST) memory hierarchy cache cache hit / miss cache line spatial locality temporal locality graphics processing unit (GPU) video memory (VRAM) ADC / DAC thermal throttling power supply unit (PSU) input / output device pointing device polling rate resolution pixel density (PPI) aspect ratio refresh rate response time backlight hard disk drive (HDD) solid-state drive (SSD) platter seek time rotational latency sequential read backup compatibility bandwidth backward compatible

1.2.13 Test Yourself

1. How long does one clock cycle take on a 2.5 GHz processor? If each instruction takes 5 cycles on average, how many instructions does the processor execute per second?

   Answer

   One cycle is $1 \div (2.5 \times 10^9)$ s $= 0.4 \times 10^{-9}$ s = 0.4 ns. 2.5×10^9 cycles are made per second and each instruction takes 5 cycles: $2.5 \times 10^9 \div 5 = 500$ million instructions/s.

2. Ten independent, indivisible jobs, each taking 3 seconds on a single core, are run on a 4-core

processor. What are the total time and the speedup over a single core?

Answer

The number of rounds is 10 ÷ 4 = 2.5 rounded up: 3 rounds. In the first two rounds 4 jobs run each, in the last round 2. The total time is 3 × 3 = 9 s. On a single core it would have taken 10 × 3 = 30 s; the speedup is 30 ÷ 9 ≈ 3.3 times, not 4.

3. An array of 12 integers (4 bytes each) is summed in order. The cache line is 32 bytes, the cache starts empty; a hit takes 1 ns and a miss 80 ns. How many misses occur and what is the total time?

Answer

One line holds 32 ÷ 4 = 8 elements. a[0]…a[7] are in the first block and a[8]…a[11] in the second. There is a miss at a[0] and a[8], and the remaining 10 accesses hit: 2 × 80 + 10 × 1 = 170 ns. Without a cache it would have taken 12 × 80 = 960 ns.

4. L1 ≈ 1 ns, L2 ≈ 3 ns, L3 ≈ 10 ns, RAM ≈ 80 ns. At the start, L1 holds A, L2 holds B, and L3 holds B and C. The processor reads C, B, C in turn; data that is found is copied to the upper levels. How long does each read take?

Answer

C: not in L1 (1), not in L2 (+3), in L3 (+10) → 14 ns; C is copied to L2 and L1. B: not in L1 (1), in L2 (+3) → 4 ns; B is copied to L1. C: now in L1 → 1 ns. The total is 19 ns; if all had been read from RAM it would have been 240 ns.

5. A 1-minute uncompressed recording is made in stereo at 48,000 samples/s and 24 bits (3 bytes) per sample. How many bytes and about how many MiB does it take?

Answer

Per second 48,000 × 3 × 2 = 288,000 bytes; in 60 seconds 288,000 × 60 = 17,280,000 bytes. 17,280,000 ÷ 1,048,576 ≈ 16.5 MiB.

6. What is the PPI of a 15.6-inch laptop screen with a resolution of 2560 × 1440? If this screen is 165 Hz, how many ms does a frame stay on the screen?

Answer

$\sqrt{(2560^2 + 1440^2)} = \sqrt{8{,}627{,}200} \approx 2937.2$; 2937.2 ÷ 15.6 ≈ 188.3 PPI. The frame duration is 1000 ÷ 165 ≈ 6.1 ms.

7. An HDD spinning at 5400 rpm has an average seek time of 12 ms and a transfer rate of 120 MB/s.

What is the average access time of a 4 KB sector? (1 MB = 1000 KB)

Answer

One turn is 60,000 ÷ 5400 ≈ 11.11 ms; the rotational latency, half a turn, is ≈ 5.56 ms. The transfer is 0.004 MB ÷ 120 MB/s ≈ 0.033 ms. The total is 12 + 5.56 + 0.03 ≈ 17.6 ms.

8. About how many GiB does a disk labeled “2 TB” on the box show in an operating system that computes with base 1024?

Answer

2 TB = 2×10^{12} bytes. $2 \times 10^{12} \div 2^{30} \approx 1862.6$ GiB. The byte count is the same; the difference comes only from the unit.

9. In a system the processor draws 65 W, the graphics card 160 W, and the other components 50 W. A 30% margin is to be left; models of 300, 400, 450, and 550 W are available. Which is chosen?

Answer

The total is 65 + 160 + 50 = 275 W; with the margin 275 × 1.3 = 357.5 W. The smallest model that does not fall below this is 400 W.

10. A program file sits on the SSD. When the program is opened and a calculation is done, through which devices, in which order, does the data pass? What happens when the result is saved?

Answer

The program is copied from the SSD to RAM; the CPU brings instructions and data from RAM into the cache and from there into the registers and processes them: SSD → RAM → cache → registers. The result goes through the graphics card to the monitor. When saved, the data is written from RAM to the SSD; until it is saved, the changes are only in RAM.

1.2.14 Further reading

- C. Petzold, *Code: The Hidden Language of Computer Hardware and Software* (2nd ed., 2022): starting from binary numbers and logic gates, it explains step by step how a processor is built, without requiring background knowledge.
- D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition* (2nd ed., 2020), Chapters 1 and 5: treat in detail how performance is calculated from clock speed and instruction count, the memory hierarchy, and cache design.

- R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective* (3rd ed., 2015), Chapter 6: covers storage technologies, the calculation of HDD access time, and the effect of locality on program speed from the programmer's viewpoint.

1.2.15 References

1. D. A. Patterson, J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware/Software Interface*, 2nd ed., Morgan Kaufmann, 2020.
2. W. Stallings, *Computer Organization and Architecture*, 11th ed., Pearson, 2019.
3. R. E. Bryant, D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3rd ed., Pearson, 2015.
4. C. Petzold, *Code: The Hidden Language of Computer Hardware and Software*, 2nd ed., Microsoft Press, 2022.
5. Technical standards of PCI-SIG (PCI Express), USB-IF (USB), VESA (DisplayPort), and JEDEC (DDR memory).

PART I · CHAPTER 1.3

Computers

A computer is built from billions of tiny switches that can tell only two states apart. Yet the same machine calculates, writes text, shows pictures, and plays music. These notes explain how that gap is bridged. First they cover how numbers are written as bit sequences: bases, negative numbers, overflow, and floating-point numbers. Then they show how letters are turned into numbers with ASCII, Unicode, and UTF-8. The notes go on with adders built from logic gates and follow, step by step, how a processor runs a program in memory with the fetch–decode–execute cycle. The last sections cover the basic design choices of computer architecture and the path of a single line of Java down to machine instructions.

LEARNING OUTCOMES

Number representation
Convert numbers among binary, octal, decimal, and hexadecimal; explain the consequences of a limited number of bits (negative numbers, overflow, floating point).

Text representation
Distinguish the concepts of ASCII, Unicode, and strings; find the UTF-8 bytes of a character by hand.

Program execution
Explain the stored-program concept and the fetch–decode–execute cycle; trace a small machine program step by step.

Architecture
Interpret the relationship among CPU, memory, registers, and buses; compare the von Neumann and Harvard architectures and the RISC and CISC approaches.

1.3.1 Computers and information

A calculator only calculates; a music player only plays music. A computer, with the same hardware, can do all of these and much more. This flexibility has two sources. The first is that every kind of information can be turned into numbers, and numbers into bits. The second is that the instructions

telling the machine what to do also sit in memory as numbers. This section introduces the main parts of the computer and the idea of turning information into bits.

1.3.1.1 The basic parts of a computer

A desktop computer, a phone, and the control board of a washing machine look very different at first glance. Yet all three contain the same three basic parts.

- The **central processing unit** (CPU), or simply the processor, executes instructions. It does jobs such as adding, comparing, and moving a value from one place to another.
- **Memory** stores both the data to be processed and the instructions the processor will execute. In these notes the word "memory" means main memory (RAM) unless stated otherwise.
- **Input and output devices** (I/O) let the computer exchange data with the outside world. The keyboard, mouse, microphone, and temperature sensor are input devices; the screen, printer, and speaker are output devices.

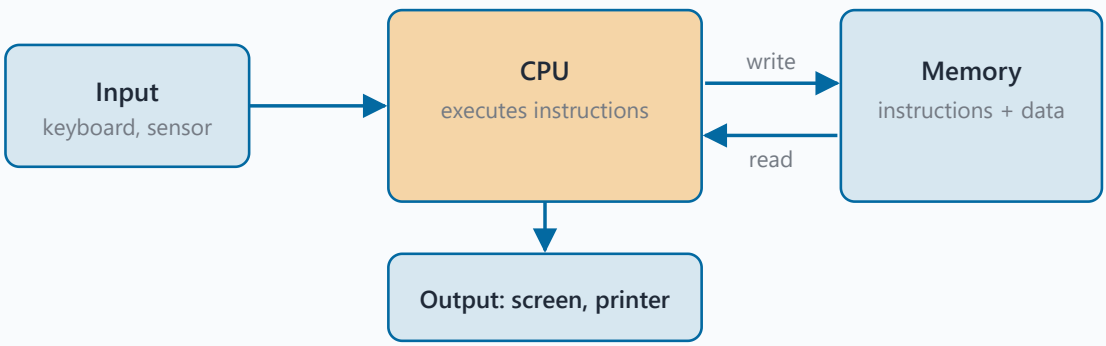


Figure 1.3.1: The basic parts of a computer. Input devices give data to the CPU. The CPU writes values to memory and reads values from memory. The result goes to an output device.

Figure 1.3.1 shows the flow of data among the parts. A key pressed on the keyboard reaches the CPU through the input device. The CPU can write this value to memory, read it back later, and operate on it. The resulting output is sent to the screen. The two arrows between the CPU and memory show that the two are in constant conversation: as long as the processor runs, it reads instructions and data from memory and writes results to it. The parts inside the CPU and its connection to memory are treated in detail in 1.3.7.

Common mistake

Confusing memory with the disk where files are stored permanently. Main memory (RAM) is fast, but its contents are erased when the power goes off. A disk or SSD is slower but keeps its contents. A program is loaded from disk into memory before it runs; the processor can execute only the instructions in memory.

1.3.1.2 From information to bits

Every circuit inside a computer basically tells two states apart: voltage present or absent, switch open or closed. These two states are written as the digits 1 and 0. A single 0 or 1 is called a **bit** (binary digit). A group of eight bits is called a **byte**.

A single bit can express only two states. When bits are put side by side, the possibilities multiply. Two bits give 4 different sequences: 00, 01, 10, 11; three bits give 8; eight bits give $2^8 = 256$. The general rule is this: n bits can form 2^n different bit sequences. Numbers, letters, the colors of the dots in a picture, and the instructions of a program are stored by pairing them with one of these sequences.

Definition 1.3.1

An **encoding** is a set of rules that pairs the elements of a kind of information with bit sequences. A **data type** determines with which encoding a bit sequence in memory is to be read.

This definition has an important consequence: a bit sequence carries no meaning by itself. If the sequence `01000001` is read as an integer, it is 65; if it is read as an ASCII character, it is the letter "A." The meaning is given by the data type the reading program chooses.

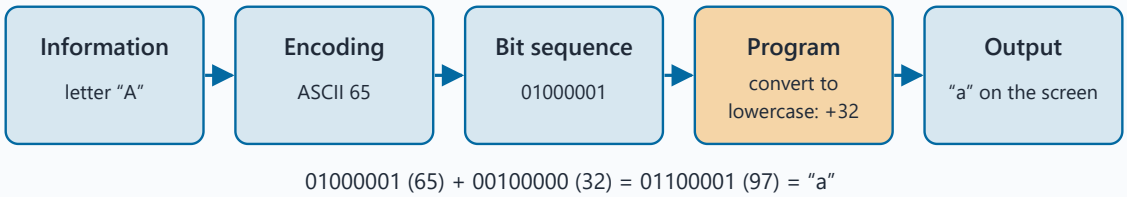


Figure 1.3.2: The journey of the letter "A." The letter is turned into the number 65 by ASCII, and 65 into the bit sequence `01000001`. The program adds 32 to this number and the result is `01100001` = 97; "a" appears on the screen.

Figure 1.3.2 shows the whole chain together. The "A" key is pressed on the keyboard. The encoding rule pairs this letter with the number 65. In memory, 65 sits as the eight-bit sequence `01000001`. If a program adds 32 to this number, it gets `01100001`, that is, 97. The screen reads this number with the same encoding and draws the letter "a." The computer does not know how to "make a letter smaller"; it only adds 32 to a number. The meaning is set up by the encoding and the data type. This small example is taken up again with bit operations in 1.3.4.2.

1.3.2 Representing integers

A computer's most basic job is to deal with numbers. This section explains how integers are written with bits. First, how the everyday decimal system works is recalled; the same idea is carried over to the binary system. Then negative numbers and the overflow problem caused by a limited number of bits are covered.

1.3.2.1 Positional number systems and the base

When 345 is written in the decimal system, each of the three digits carries a different value. The 5 is in the ones position, the 4 in the tens position, and the 3 in the hundreds position:

$$345 = 3 \cdot 100 + 4 \cdot 10 + 5 \cdot 1 = 3 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0$$

The value of a digit changes with its position. This is why such systems are called **positional number systems**. The weight of each position is 10 times the weight of the position to its right. This 10 is called the **base** (or radix) of the system.

Definition 1.3.2

A positional system with base b uses b digits, from 0 to $b - 1$. Of the positions numbered 0, 1, 2, … from right to left, position i has weight b^i. The value of the number is the sum of each digit multiplied by its own weight.

In computer science four bases are in common use. Table 1.3.1 shows the same number, 13, in all of them. To avoid confusion about which base a number is written in, the base is added to the lower right in small digits: 1101_2, 15_8, 13_{10}, D_{16}.

System	Base	Digits	Example	Value
Binary	2	0–1	1101_2	$1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 13$
Octal	8	0–7	15_8	$1 \cdot 8 + 5 \cdot 1 = 13$
Decimal	10	0–9	13_{10}	$1 \cdot 10 + 3 \cdot 1 = 13$
Hexadecimal	16	0–9, A–F	D_{16}	$13 \cdot 1 = 13$

Table 1.3.1: Commonly used number systems. All four examples show the same number (thirteen).

The hexadecimal system needs sixteen digits, but everyday writing has only ten. The letters A–F are used for the six missing digits: A = 10, B = 11, C = 12, D = 13, E = 14, F = 15. In programming languages the base is often marked with a prefix. In Java, `0b1101` is a binary constant and `0xD` a hexadecimal one.

1.3.2.2 Decimal to binary, binary to decimal

Converting binary to decimal is applying the definition directly. In the binary system the position weights double from right to left: 1, 2, 4, 8, 16, 32, 64, 128, …. The weight of every position that holds a 1 is added to the sum. For example, in 101101_2 the 1s are in the 32, 8, 4, and 1 positions: 32 + 8 + 4 + 1 = 45.

The most systematic way to **convert decimal to binary** is to divide the number by 2 repeatedly. The remainder of each division (0 or 1) is a bit. The process stops when the quotient is 0. The remainders are read from last to first, because the first remainder is the rightmost, that is, the ones position.

Example 1.3.1

Let us convert 45 to binary.

Dividend	÷ 2 quotient	Remainder (bit)	Position weight
45	22	1	1
22	11	0	2
11	5	1	4
5	2	1	8
2	1	0	16
1	0	1	32

The remainders are read from bottom to top: $45_{10} = 101101_2$. Check: 32 + 8 + 4 + 1 = 45.

For small numbers a second method is more practical: subtract the largest power of two that fits. The largest power that fits into 45 is 32; the remainder is 13. 8 fits into 13, leaving 5; 4 fits into 5, leaving 1; 1 fits into 1. The powers used are 32, 8, 4, 1, and a 1 is written in these positions and a 0 in the others: 101101.

Java's ready-made methods do the same conversions. The code below writes 45 in three bases, then converts a binary string back to a number. The last line prints the sum of two constants written with the `0b` and `0x` prefixes; both are 45, so the result is 90.

```
int n = 45;
System.out.println(Integer.toBinaryString(n));
System.out.println(Integer.toOctalString(n));
System.out.println(Integer.toHexString(n));
System.out.println(Integer.parseInt("101101", 2));
System.out.println(0b101101 + 0x2D);
```

Output:

```
101101
55
2d
45
90
```

Because Java writes hexadecimal digits in lowercase, 45 appears as "2d." $2D_{16}$ and $2d_{16}$ are the same number.

1.3.2.3 Octal and hexadecimal: grouping bits

Binary notation is natural for the computer but long and hard to read for people. Writing and reading a 32-bit number with thirty-two digits invites mistakes. The octal and hexadecimal bases are ways of writing long binary numbers briefly. The reason these two bases are chosen is simple: $8 = 2^3$ and $16 = 2^4$.

Thanks to these equalities, no arithmetic is needed for the conversion. The bits are grouped starting from the right: in threes for octal, in fours for hexadecimal. Each group is converted to a single digit. If the leftmost group is incomplete, a 0 is added on its left; leading zeros do not change the value.

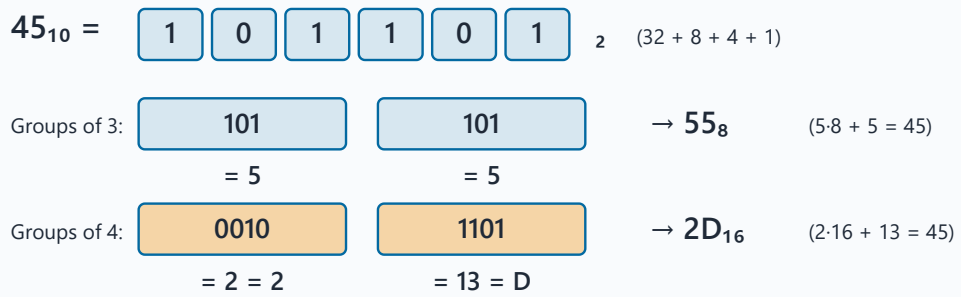


Figure 1.3.3: Converting $45_{10} = 101101_2$ to octal and hexadecimal by grouping bits. The groups of three are 101 and 101: 55_8. The groups of four are 0010 and 1101: $2D_{16}$.

In Figure 1.3.3, 101101 is first grouped in threes: 101 | 101. Each group is 5 and the result is 55_8. Check: $5 \cdot 8 + 5 = 45$. When grouping in fours, the group on the right is 1101; on the left only two bits remain (10), and two 0s are added in front to make 0010. Since 0010 = 2 and 1101 = 13 = D, the result is $2D_{16}$. Check: $2 \cdot 16 + 13 = 45$.

The reverse direction is just as easy. To convert a hexadecimal number to binary, each digit is written with four bits. For example, in $A7_{16}$, A = 1010 and 7 = 0111; the result is 10100111_2. Its decimal value is $10 \cdot 16 + 7 = 167$. Because of this convenience, memory addresses, color codes, and character codes are mostly written in hexadecimal. A byte always corresponds to exactly two hexadecimal digits.

Example 1.3.2

Let us write 156 in binary, octal, and hexadecimal.

1. **Binary:** The powers that fit into 156 are 128, 16, 8, and 4 (128 + 16 + 8 + 4 = 156). A 1 is written in these positions: 10011100_2.
2. **Octal:** The groups of three from the right are 100, 011, and 10. The last group is made 010. The groups are 2, 3, 4, so the result is 234_8. Check: $2 \cdot 64 + 3 \cdot 8 + 4 = 156$.
3. **Hexadecimal:** The groups of four are 1001 and 1100, that is, 9 and 12 (C). The result is $9C_{16}$. Check: $9 \cdot 16 + 12 = 156$.

Common mistake

Reading a number whose base is not written as decimal. "1101" can be one thousand one hundred one, or it can be thirteen. To leave the base unambiguous, a subscript (1101_2) or a prefix (`0b1101` , `0x1101`) is used. Similarly, 10_{16} is not ten but 16; in hexadecimal, ten is written with A.

1.3.2.4 A limited number of bits and unsigned numbers

On paper the number of digits in a number is unlimited; as many digits as needed are added. In a computer the situation is different. The small, fast storage cells inside the processor are called **registers**, and the number of bits in each register is fixed: 8, 16, 32, or 64. Every variable in memory also takes a certain number of bits. In Java, `byte` is 8, `short` 16, `int` 32, and `long` 64 bits.

If all the bits are used for the magnitude of the number, an **unsigned** number results. With n bits, the numbers from 0 to $2^n - 1$ can be written. In 8 bits the largest number is $11111111_2 = 255$. Negative numbers cannot be written this way. To write them, one of the bits must be given the job of a "sign."

1.3.2.5 Negative numbers and two's complement

The first way that comes to mind for writing negative numbers is to set aside the leftmost bit for the sign: 0 for plus, 1 for minus. This method is called sign–magnitude representation, and it has two flaws. First, zero gets two separate writings: 00000000 (+0) and 10000000 (−0). Second, the addition circuit has to look at the signs and do different things for different cases. For this reason modern computers usually store negative integers in **two's complement** representation.

Definition 1.3.3

In n-bit **two's complement** representation, the weight of the leftmost bit is -2^{n-1} instead of $+2^{n-1}$; the weights of the other bits do not change. So if the leftmost bit is 1 the number is negative, and this bit is called the **sign bit**. The representable range is from -2^{n-1} to $2^{n-1} - 1$; in 8 bits, −128 ... 127.

The definition may look abstract at first reading. An example makes it concrete. In 8 bits, the value of the sequence 11111011 is found by taking the weight of the leftmost bit as −128: −128 + 64 + 32 + 16 + 8 + 2 + 1 = −5. If the same sequence were read as unsigned, it would be 128 + 64 + 32 + 16 + 8 + 2 + 1 = 251. The difference between the two readings is exactly 256, that is, 2^8. This relationship holds for every negative number: in 8 bits, the bit sequence of $-k$ is the same as the bit sequence of unsigned $256 - k$.

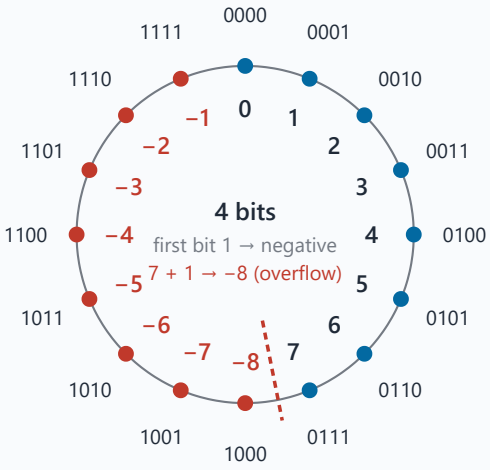


Figure 1.3.4: The 4-bit two's complement circle. The sequences from 0000 to 0111 are the positive numbers 0 ... 7; the sequences from 1000 to 1111 are the negative numbers −8 ... −1. Every sequence whose first bit is 1 is negative. Adding 1 to 7 moves to −8 on the circle: this is an overflow.

Figure 1.3.4 lays out all the 4-bit sequences like a clock face. Moving clockwise means adding 1. Starting at 0000 and adding 1 each time reaches 1, 2, ..., 7. Adding 1 once more to 0111 gives 1000, and this sequence is −8. The dial thus shows a "cliff": right after the largest positive number comes the smallest negative one. Going counterclockwise, one moves from 0000 to 1111, that is, to −1. It is therefore not surprising that all the bits of −1 are 1.

Two's complement has three important advantages.

- **A single zero.** Zero has only one writing (0000); there is no distinction between plus zero and minus zero.

- **Common arithmetic.** The addition circuit uses the same rules for positive and negative numbers. It does not look at the sign of the number; it adds the bits with ordinary binary addition.
- **Easy negation.** The negative of a number is found in two simple steps (1.3.2.6).

Bits	Signed range	Unsigned range	Java type
8	−128 … 127	0 … 255	`byte`
16	−32,768 … 32,767	0 … 65,535	`short`
32	−2,147,483,648 … 2,147,483,647	0 … 4,294,967,295	`int`
64	about $\pm 9.22 \cdot 10^{18}$	0 … about $1.84 \cdot 10^{19}$	`long`

Table 1.3.2: Signed (two's complement) and unsigned ranges by number of bits. The corresponding Java types are in the last column.

Note that the range is not symmetric. In 8 bits −128 can be written but +128 cannot. The reason is that one of the 256 sequences is set aside for zero. The remaining 255 sequences are shared between 128 negative and 127 positive numbers.

1.3.2.6 How is −5 written?

Two steps are enough to find the negative of a number in two's complement:

1. All the bits are flipped (0 ↔ 1). The resulting sequence is called the **one's complement**.
2. 1 is added to the result.

Example 1.3.3

Let us write −5 in 8 bits and check the result.

1. **Write +5:** 5 = 4 + 1, so +5 = 00000101.
2. **Flip the bits:** 00000101 → 11111010. This sequence is the one's complement and is not yet −5.
3. **Add 1:** 11111010 + 1 = 11111011. The leftmost 1 shows that the number is negative.
4. **Check:** 5 + (−5) should be 0. 00000101 + 11111011 = 100000000. The result is 9 bits; the 9th bit does not fit in an 8-bit register and is discarded. What remains is 00000000 = 0.

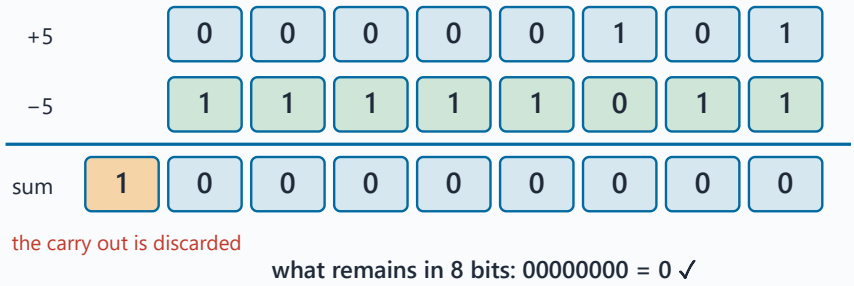


Figure 1.3.5: The 8-bit sum of +5 and −5. The sum is 9 bits long. The carried-out 9th bit (the carry) is discarded, and what remains in 8 bits is 00000000 = 0.

The check in Figure 1.3.5 shows the real strength of two's complement. The addition circuit does not know that −5 is negative. It adds the bits as usual, discards the carried-out bit, and gets the right result. No separate circuit is needed for negative numbers.

The same two steps also work in the opposite direction. To find the magnitude of a negative sequence, the bits are again flipped and 1 is added: 11111011 → 00000100 → 00000101 = 5; so the sequence is −5. In Java the tilde (`~`) is the operation that flips all the bits, so `~5 + 1` is −5:

```
byte x = -5;
System.out.println(Integer.toBinaryString(x & 0xFF));
System.out.println(~5 + 1);
System.out.println(Byte.MIN_VALUE + " " + Byte.MAX_VALUE);
```

Output:

```
11111011
-5
-128 127
```

The `x & 0xFF` in the first line is there to take only the last 8 bits of the `byte` value. That way the sequence of −5 in memory appears as 11111011. The last line gives the limits of the `byte` type, as in Table 1.3.2.

Common mistake

Flipping the bits but forgetting to add 1 in two's complement. 11111010 is the one's complement and in two's complement representation it is −6, not −5. After flipping the bits, 1 must always be added: 11111011 = −5.

1.3.2.7 Subtraction is really addition: 5 − 9

Another consequence of two's complement is that the processor needs no separate subtraction circuit. Subtraction is turned into addition by this equality:

> a − b = a + (−b)

First the two's complement of *b* is found, and then it is added to *a*. The steps in the slides are written in the pseudocode below. To the right of each line, after the semicolon, is its explanation.

```
a ← 00000101      ; +5
b ← 00001001      ; +9
inv ← NOT b       ; flip the bits
neg ← inv + 1     ; −9
s ← a + neg       ; 5 + (−9)
print(s)          ; read as signed
```

Line	Variable	Bits	Explanation
1	`a`	`00000101`	+5
2	`b`	`00001001`	+9 = 8 + 1
3	`inv`	`11110110`	all bits flipped
4	`neg`	`11110111`	1 added: −9 (247 = 256 − 9 if read as unsigned)
5	`s`	`11111100`	5 + 247 = 252; it does not reach 256, so there is no carry
6	printed	−4	sign bit 1, magnitude: flip + 1 → 00000100 = 4

Table 1.3.3: Trace of the operation 5 − 9. Each row shows the state after one line of the pseudocode has been executed.

The last row of the table shows how the result is read. Because the leftmost bit of `s` is 1, the number is negative. To find its magnitude, the bits are flipped again (00000011) and 1 is added (00000100 = 4). So *s* = *−4*, and 5 − 9 = −4 is correct.

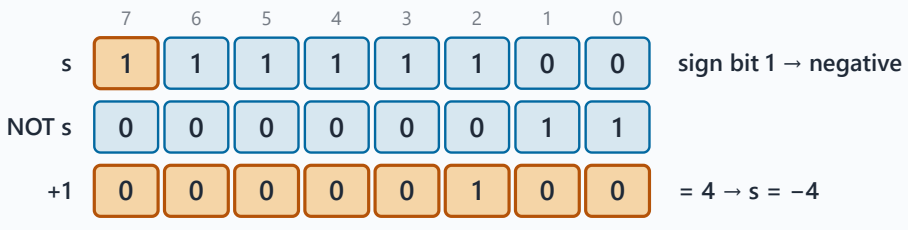


Figure 1.3.6: Reading the result of 5 − 9. The sign bit of `s` = 11111100 is 1, so the number is negative. Flipping the bits and adding 1 gives 00000100 = 4; *s* = *−4*.

There is a remarkable point here: the processor produces the same sequence 11111100 and does not know how to interpret it. Read as unsigned, this sequence is 252; read as signed, it is −4. The program decides which is correct by looking at the variable's data type.

1.3.2.8 Overflow: 200 + 100 in 8 bits

A limited number of bits means that the result of some operations does not fit in the register. An 8-bit unsigned register holds at most 255. What happens if 200 and 100 are added?

Example 1.3.4

1. **Operands:** 200 = 128 + 64 + 8 → 11001000; 100 = 64 + 32 + 4 → 01100100.
2. **True sum:** 300 = 256 + 32 + 8 + 4 → 100101100_2. The result needs 9 bits.
3. **What stays in the register:** The register holds only 8 bits. The 9th bit, with weight 256, is lost and 00101100 = 44 is left.
4. **Shortcut:** 300 − 256 = 44.

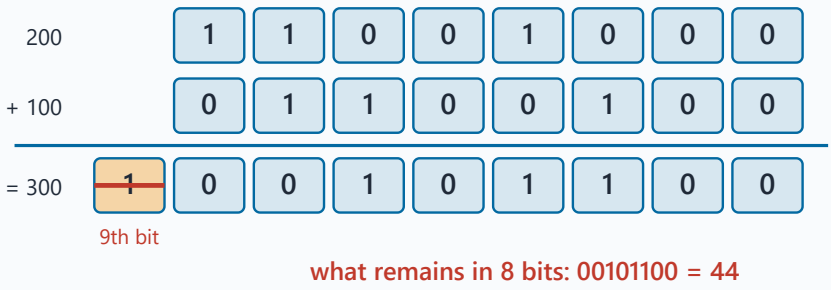


Figure 1.3.7: 200 + 100 in 8 bits. The true sum 300 = 100101100_2 needs 9 bits. The 9th bit (256) does not fit in the register; what remains in 8 bits is 00101100 = 44.

Definition 1.3.4

Overflow is when the true result of an arithmetic operation does not fit in the range of the register or variable that is to hold it. When overflow occurs, the high bits that do not fit are discarded and a wrong value is left in the register.

Overflow occurs with signed numbers too. If 127 + 1 is computed in 8 bits, 01111111 + 1 = 10000000 results, and this sequence is −128. The move from 7 to −8 on the dial in Figure 1.3.4 is the 4-bit counterpart of the same event.

Overflow is a silent error in programs. Java gives no error on integer overflow; the result silently comes out wrong:

```
byte b = (byte) (100 + 50);
System.out.println(b);
int big = Integer.MAX_VALUE;
System.out.println(big);
System.out.println(big + 1);
```

Output:

```
-106
2147483647
-2147483648
```

In the first line 100 + 50 = 150 is computed and the result is squeezed into 8 bits with `(byte)`. 150 does not fit the `byte` range; its 8-bit sequence is 10010110, and when this sequence is read as signed it is 150 − 256 = −106. In the following lines 1 is added to the largest value of `int` and the smallest value results. So when choosing the type of a variable, the largest value it may take should be considered. For example, a country's population does not fit in a `short`; the world's population does not even fit in an `int` (about 8 billion > 2.1 billion).

1.3.2.9 Carry versus signed overflow

The addition circuit can produce two separate warnings, and the two are often confused. The first is the **carry** out of the leftmost position, that is, the 9th bit. The second is **signed overflow**. The two are problems of different interpretations.

- **Carry** shows that, when the numbers are interpreted as *unsigned*, the result does not fit in 8 bits. In signed addition the carry is discarded and the result can still be correct.
- **Signed overflow** occurs when the sum of two numbers of the same sign comes out with the opposite sign. If the sum of two positives looks negative, or the sum of two negatives looks positive, the result is wrong. The sum of two numbers of different signs never overflows, because the magnitude of the sum cannot exceed the larger of the two numbers.

Operation	Binary sum	Carry	Signed overflow	Result
5 + (−9)	`00000101 + 11110111 = 11111100`	0	none	−4 ✓
(−1) + (−1)	`11111111 + 11111111 = 1 11111110`	1	none	−2 ✓
100 + 50	`01100100 + 00110010 = 10010110`	0	yes	−106 ✗
(−100) + (−50)	`10011100 + 11001110 = 1 01101010`	1	yes	+106 ✗

Table 1.3.4: Four additions in 8 bits. Carry and signed overflow occur independently of each other.

Table 1.3.4 shows all four possible cases. In the first row there is neither carry nor overflow. In the second row there is a carry, but the result is correct: −1 + (−1) = −2. A carry is a problem only in the unsigned interpretation; read as unsigned, this operation is 255 + 255 = 510, which does not fit in 8 bits. In the third row the sum of two positive numbers has come out negative; there is no carry but there is signed overflow. In the fourth row the sum of two negatives has come out positive; there is both a carry and an overflow.

Example 1.3.5

In 8 bits, is there a carry and signed overflow in the operations 120 + 10 and (−3) + (−5)?

1. **120 + 10:** 01111000 + 00001010 = 10000010. Because the sum fits in 8 bits, the carry is 0. But the sign bit of the sum of two positive numbers has come out 1: there is signed overflow. Read as signed, the sequence is −126; the correct result, 130, does not fit in the 8-bit signed range.
2. **(−3) + (−5):** −3 = 11111101, −5 = 11111011. The sum is 1 11111000; the carry is 1 and is discarded. What remains, 11111000, is negative and its magnitude is 00001000 = 8: the result is −8, which is correct. Because the sum of two negatives came out negative, there is no signed overflow.

Common mistake

Taking carry and signed overflow to be the same thing. Carry is a problem of the unsigned interpretation, overflow of the signed one. One can occur without the other: in (−1) + (−1) there is a carry but the result is correct; in 100 + 50 there is no carry but the signed result is wrong.

1.3.3 Floating-point numbers

Integers are enough for counting, but most engineering calculations are done with fractional values: the voltage across a resistor may be 3.3 V, the period of a signal 0.000016 s, the distance to a star $4 \cdot 10^{16}$ m. This section explains how such numbers are stored in a limited number of bits and what this costs.

1.3.3.1 Why floating point?

In scientific notation, very large and very small numbers are written in the same pattern: $4 \cdot 10^{16}$, $1.6 \cdot 10^{-5}$. The number is split into two parts: the significant digits and the exponent that says where the point is. As the exponent changes, the point “floats” left or right. The **floating-point** representation in computers applies the same idea with base 2 instead of 10.

This representation has four properties.

- **Wide range.** Very large and very small values are stored with the same number of bits. A 32-bit `float` can hold values between about $1.2 \cdot 10^{-38}$ and $3.4 \cdot 10^{38}$.
- **Limited precision.** The number of significant digits is fixed. A `float` carries about 7 decimal digits and a `double` about 15–16. Not every decimal value can be represented exactly in a finite number of bits.
- **Rounding.** Digits that cannot be stored are rounded; small differences can appear in the results of calculations.
- **Comparison.** Because of these small differences, floating-point results are compared not with exact equality but with a tolerance value.

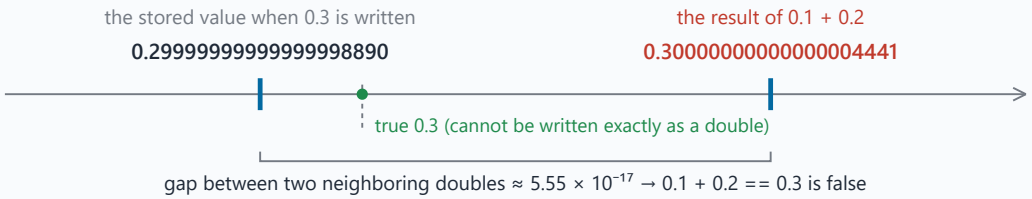


Figure 1.3.8: The number line around 0.3. When 0.3 is written in a program, the stored `double` value is 0.29999999999999998890. The result of 0.1 + 0.2 is the next `double`: 0.30000000000000004441. The true 0.3 lies between these two and cannot be written exactly as a `double`. The distance between two neighboring `double`s is about $5.55 \cdot 10^{-17}$.

Figure 1.3.8 shows concretely what limited precision means. Not every point on the number line can be written as a `double`; only certain points can, and there are gaps between them. 0.3 is not one of these points. When a program writes 0.3, the nearest point is stored. 0.1 and 0.2 cannot be stored exactly either; when their rounded versions are added, the result falls on the next point instead of the rounded version of 0.3. This is why, in Java, `0.1 + 0.2` is printed as `0.30000000000000004` and the comparison `0.1 + 0.2 == 0.3` gives `false`.

1.3.3.2 Converting a decimal fraction to binary

In the binary system, the weights of the positions after the point halve: 1/2, 1/4, 1/8, 1/16, ... For example, 0.101_2 = 1/2 + 1/8 = 0.625. To convert a decimal fraction to binary, the reverse of the division method for integers is used.

Definition 1.3.5

Method for converting a fraction to binary: The fraction is multiplied by 2. The integer part of the product (0 or 1) is the next bit after the point. The integer part is dropped and the process continues with the remaining fraction. When the fraction becomes 0, the expansion ends. If a fraction seen before comes up again, the expansion has entered a loop and never ends.

Example 1.3.6

Let us convert 0.375 to binary.

1. $0.375 \times 2 = 0.75 \rightarrow$ integer part 0, first bit 0.
2. $0.75 \times 2 = 1.5 \rightarrow$ integer part 1, second bit 1; the process continues with the fraction 0.5.
3. $0.5 \times 2 = 1.0 \rightarrow$ integer part 1, third bit 1; the fraction became 0 and the expansion ended.

The result is $0.375 = 0.011_2$. Check: $1/4 + 1/8 = 0.25 + 0.125 = 0.375$.

Because the binary expansion of 0.375 is finite, this number is stored exactly in the computer. The reason is that 0.375 is 3/8, and its denominator is a power of 2. The general rule is this: if the denominator of the reduced fraction consists only of powers of 2, the binary expansion is finite; otherwise it repeats forever. 0.5, 0.25, 0.75, and 0.375 are stored exactly. 0.1 = 1/10, 0.2 = 1/5, and 0.3 = 3/10 cannot be stored exactly because their denominators have a factor of 5.

1.3.3.3 Why can't 0.1 be stored exactly?

Let us see step by step what happens when the same method is applied to 0.1.

Step	Operation	Bits	Remaining fraction
1	$0.1 \times 2 = 0.2$	0	0.2
2	$0.2 \times 2 = 0.4$	0	0.4
3	$0.4 \times 2 = 0.8$	0	0.8
4	$0.8 \times 2 = 1.6$	1	0.6
5	$0.6 \times 2 = 1.2$	1	0.2 (the fraction at step 2)

Table 1.3.5: The binary expansion of 0.1. After the fifth step the fraction becomes 0.2 again and the expansion enters a loop.

At the end of the fifth step the fraction is 0.2 again. 0.2 was already processed at the second step; from here on steps 2–5 repeat exactly and the bits 0011 are produced forever:

> $0.1_{10} = 0.0\ 0011\ 0011\ 0011\ \ldots_2$

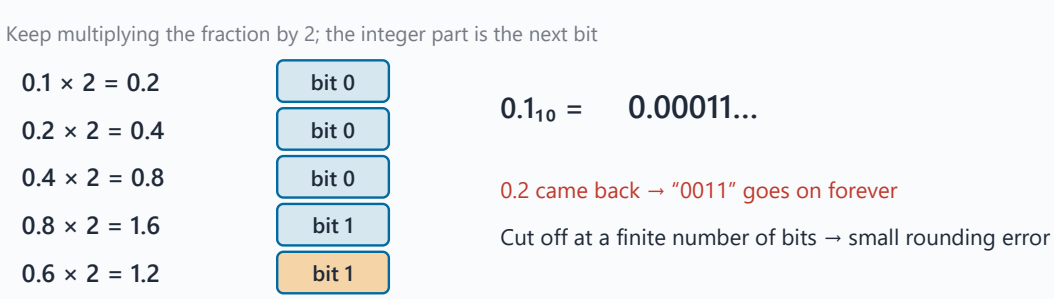


Figure 1.3.9: Converting 0.1 to binary. After five multiplications the fraction 0.2 is seen again; the pattern "0011" goes on forever. Cutting it off at a finite number of bits produces a small rounding error.

A computer cannot store infinitely many bits. The expansion is cut off at some point and the last bit is rounded. The stored number is very close to 0.1 but not equal to it. It is like writing 1/3 as 0.3333… in the decimal system: however many digits are written, 1/3 is never reached exactly. The 0.1, which causes no trouble in the decimal system, behaves like 1/3 in the binary system.

Small rounding errors can accumulate in successive operations. The code below adds 0.1 ten times:

```
double sum = 0.0;
for (int i = 0; i < 10; i++) {
    sum = sum + 0.1;
}
System.out.println(sum);
System.out.println(sum == 1.0);
System.out.println(Math.abs(sum - 1.0) < 1e-9);
```

Output:

```
0.9999999999999999
false
true
```

In mathematics $10 \cdot 0.1 = 1$, but the loop produces 0.9999999999999999. This is why the `==` comparison in the second line gives `false`. The last line shows the right method: if the absolute value of the difference of two numbers is smaller than a small threshold (here 10^{-9}), the numbers are taken to be "equal."

Common mistake

Comparing floating-point results with `==`. Because 0.1 + 0.2 ≠ 0.3, `if (x == 0.3)` may be skipped unexpectedly. The right way is to compare with a tolerance of the form `Math.abs(a - b) < epsilon`. The threshold is chosen according to the scale of the quantities being computed.

1.3.3.4 The IEEE 754 format

How floating-point numbers are laid out in bits is set by the **IEEE 754** standard, which all processors follow. The standard puts the number in binary scientific form:

> $\text{value} = (-1)^{\text{sign}} \times 1.\text{fraction} \times 2^{(\text{exponent} - \text{bias})}$

The bit sequence is divided into three fields: **sign**, **exponent**, and **fraction**. Table 1.3.6 shows the fields of the two floating-point types in Java.

Type	Total	Sign	Exponent	Fraction	Bias	Decimal digits
`float`	32 bits	1	8	23	127	about 7
`double`	64 bits	1	11	52	1023	about 15–16

Table 1.3.6: IEEE 754 formats. The bias is a constant added to the true exponent before it is written to the exponent field.

The roles of the fields are as follows.

- If the **sign bit** is 0 the number is positive, if 1 negative. Two's complement is not used in floating-point numbers; the magnitude is always stored as positive.
- The **exponent field** holds not the true exponent but the exponent with the **bias** added. In `float` the bias is 127. That way negative exponents are also stored as an unsigned number: if the exponent is −4, 123 is written to the field.
- The **fraction field** holds the bits after the point. In normal numbers there is always a single 1 to the left of the point. Because this 1 never changes, it is not stored; the standard gains a bit.

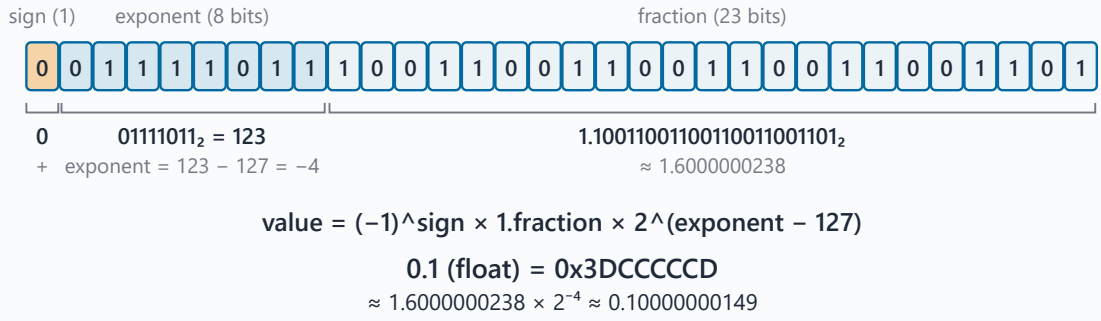


Figure 1.3.10: The 32-bit `float` representation of 0.1. The sign is 0; the exponent field is $01111011_2 = 123$, so the true exponent is $123 - 127 = -4$; the fraction is 1001 1001 … 1101. The value is $1.6000000238 \times 2^{-4} \approx 0.10000000149$ and its hexadecimal form is 3DCCCCCD.

Figure 1.3.10 shows how 0.1 is stored as a `float`. $0.1 = 1.1001\ 1001\ 1001\ldots_2 \times 2^{-4}$. The pattern 1001 in the fraction repeats forever. When it does not fit in 23 bits, the last bit is rounded; this is why 1101 appears at the end of the fraction field instead of 1100. The stored value is exactly 0.100000001490116119384765625: very close to 0.1, but not equal. Java can show this value with all its digits using `BigDecimal`:

```
System.out.println(Integer.toHexString(Float.floatToIntBits(-6.25f)));
System.out.println(Integer.toHexString(Float.floatToIntBits(0.1f)));
System.out.println(new java.math.BigDecimal(0.1f));
```

Output:

```
c0c80000
3dcccccd
0.100000001490116119384765625
```

The `Float.floatToIntBits` method copies the 32 bits of a `float` as they are into an `int`; `Integer.toHexString` then writes these bits in hexadecimal. The c0c80000 in the first line will be found by hand in the next section.

1.3.3.5 Encoding −6.25 as a float

The best way to understand IEEE 754 is to encode a number by hand. The steps in the slides are summarized in the pseudocode below:

```
x ← -6.25
sign ← 1          ; negative
bin ← 110.01       ; 6 = 110, 0.25 = 0.01
norm ← 1.1001 × 2² ; point 2 left
exp ← 2 + 127 = 129 ; 10000001
frac ← 1001 0000… ; leading 1 dropped
print(hex(bits))
```

Example 1.3.7

Let us find the 32-bit `float` form of −6.25.

1. **Sign:** Because the number is negative, the sign bit is 1. The remaining steps are done on 6.25.
2. **Converting to binary:** The integer part is $6 = 110_2$. For the fraction 0.25: $0.25 \times 2 = 0.5 \rightarrow 0$; $0.5 \times 2 = 1.0 \rightarrow 1$ and the fraction drops to 0: 0.01_2. So $6.25 = 110.01_2$. Unlike 0.1, the expansion is finite and the number is stored exactly.
3. **Normalization:** The point is moved until a single 1 is left to its left. In 110.01 the point moves 2 positions to the left: 1.1001×2^2. The true exponent is 2.
4. **Exponent field:** The bias is added: $2 + 127 = 129 = 10000001_2$.
5. **Fraction field:** The leading 1 is not written; only the 1001 after the point is written. The remaining 19 bits are filled with 0: 1001 0000 0000 0000 0000 000.
6. **Combining the bits:** 1 | 10000001 | 10010000000000000000000. The 32 bits grouped in fours: 1100 0000 1100 1000 0000 0000 0000 0000 → $C0C80000_{16}$.

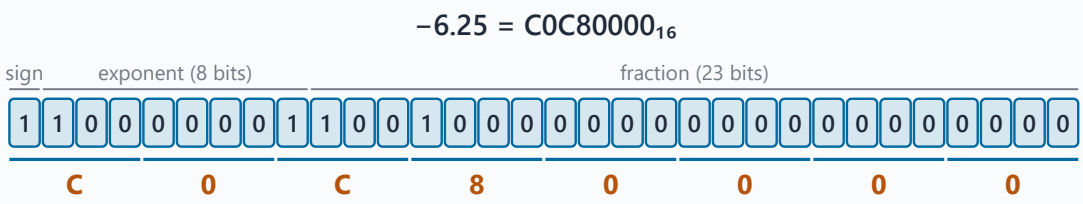


Figure 1.3.11: The 32 bits of −6.25. The sign is 1, the exponent 10000001 (129), the fraction 1001 and nineteen 0s. The groups of four are C, 0, C, 8, 0, 0, 0, 0: $-6.25 = C0C80000_{16}$.

The C0C80000 found by hand is the same as Java's result for `Float.floatToIntBits(-6.25f)` (c0c80000). Reading in the reverse direction is also the reverse of the same steps. For example, the bits of 41280000_{16} are 0 | 10000010 | 0101000…. The sign is 0, the true exponent is 130 − 127 = 3, and the number is $1.0101_2 \times 2^3 = 1010.1_2 = 10.5$.

Common mistake

Writing the true exponent in the exponent field, or putting the leading 1 into the fraction field as well. In the −6.25 example, 129 (10000001) is written to the exponent field, not 2 (00000010). The fraction field begins with 1001, not 11001; the 1 to the left of the point is not stored.

1.3.3.6 Working with floating-point numbers

Knowing the limits of floating-point numbers turns into three practical rules in everyday programming.

- **Tolerance instead of equality.** Two computed values are not compared with `==`; the absolute value of their difference is compared with a small threshold.
- `double` **for precision.** About 7 digits of `float` are too few for most engineering calculations. In Java, fractional constants are `double` by default, and it is usually the one chosen.
- **Integers for money.** In calculations where an error of a kuruş is not acceptable, amounts are kept as integers in kuruş, or in a type that does exact decimal arithmetic (`BigDecimal` in Java).

The difference between integers and floating-point numbers becomes clear here. Binary representation is exact for every integer that fits in the range: 13 is stored as 13. Floating point, on the other hand, is approximate for some decimal values. Overflow is the typical problem of integers and rounding that of floating-point numbers; both come from the limited number of bits.

1.3.4 Representing text

A computer cannot store letters directly; it stores only numbers. To store text, every character is given a number, and this number is written with bits. Which number goes to which character is decided by an encoding table. This section starts from ASCII, the oldest and simplest table, and reaches Unicode, which covers the writing systems of the whole world, and UTF-8, which writes it into bytes.

1.3.4.1 ASCII

ASCII (American Standard Code for Information Interchange) pairs the basic Latin letters, digits, punctuation marks, and control characters with numeric codes. It is a 7-bit code defined in the 1960s and contains $2^7 = 128$ characters (0–127). When stored in a byte, the highest bit is 0.

Character	Decimal	Hexadecimal	Binary (8 bits)
“A”	65	41	`01000001`
“Z”	90	5A	`01011010`
“a”	97	61	`01100001`

Character	Decimal	Hexadecimal	Binary (8 bits)
“z”	122	7A	`01111010`
“0”	48	30	`00110000`
“7”	55	37	`00110111`
space	32	20	`00100000`
line end (LF)	10	0A	`00001010`

Table 1.3.7: Some ASCII codes. Uppercase letters start at 65, lowercase letters at 97, and digits at 48, and each group is consecutive.

The last row of the table is a **control character**. Control characters do not appear as a shape on the screen; they say how the text is to be laid out. The line end (LF, line feed) says the cursor must move to the next line. The first 32 codes of ASCII (0–31) and 127 are of this kind.

The layout of the ASCII table is not random; two properties make some calculations easy.

- **Letters and digits are consecutive.** “B” = 66, “C” = 67, and so on. The numeric value of a digit character is found by subtracting the code of “0” from its code: “7” − “0” = 55 − 48 = 7.
- **Between an uppercase and a lowercase letter there is always a difference of 32.** “a” − “A” = 97 − 65 = 32. Since 32 = 2^5, the two letters differ in a single bit.

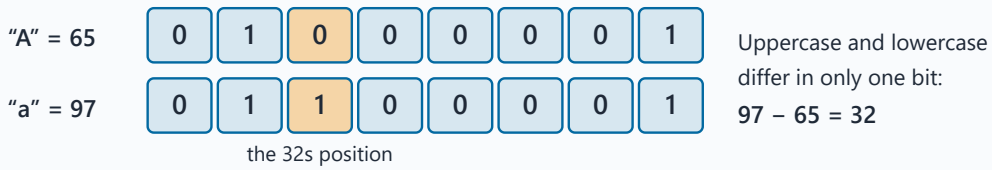


Figure 1.3.12: “A” = 65 = 01000001 and “a” = 97 = 01100001 differ only in the bit in the 32s position.

Common mistake

Confusing a digit character with a number. A “7” entered from the keyboard is a character and its code is 55. If 55 rather than 7 goes into a calculation as its numeric value, the result comes out wrong. The right thing is to do the conversion “7” − “0” = 7.

ASCII’s great shortcoming is its coverage. 128 characters are enough for English, but Turkish letters such as “ç, ğ, ı, ö, ş, ü,” the Greek and Arabic alphabets, Chinese characters, and mathematical symbols are not in the table. In the 1980s and 90s separate 8-bit tables were made for each language; the same byte meant “ğ” in one table and a different letter in another. This is mostly why Turkish texts look garbled on another computer. Unicode was developed to resolve this confusion (1.3.4.3).

1.3.4.2 Bit operations: changing the case of a letter

That uppercase and lowercase letters differ in a single bit is a good example for introducing **bitwise operations**. Bitwise operations process the bits of two numbers at the same positions one by one. To change only some bits of a number, a **bit mask** is used: the positions that are 1 in the mask take part in the operation and the 0 positions are preserved.

The difference between “G” = 71 and “g” = 103 is 32 = 2^5. So the two letters differ in bit 5 (bits are numbered from the right, starting at 0). As the mask, 00100000, which has only this bit set to 1, is chosen. The pseudocode in the slides is as follows:

```
c ← 'G'            ; 71 = 01000111
m ← 00100000       ; mask: bit 5 (32)
k ← c OR m         ; set the bit
b ← k AND (NOT m)  ; clear the bit
t ← b XOR m        ; flip the bit
s ← c << 1         ; shift left
```

Operation	Bits	Value	Rule
`c` = “G”	`01000111`	71 = “G”	start
`c OR m`	`01100111`	103 = “g”	the bit that is 1 in the mask becomes 1
`k AND (NOT m)`	`01000111`	71 = “G”	NOT m = 11011111; the bit in the mask is pulled to 0
`b XOR m`	`01100111`	103 = “g”	the bit in the mask is flipped
`c << 1`	`10001110`	142 = 71 × 2	the bits shift one position to the left, a 0 enters from the right
`c >> 1`	`00100011`	35 = 71 ÷ 2 (integer part)	the bits shift one position to the right

Table 1.3.8: Bit operations on the letter “G.” The mask `m` = 00100000 (32).

What each operation is good for can be seen in the last column of the table. **OR** surely makes a bit 1: if the letter is already lowercase it does not change, and if uppercase it becomes lowercase. **AND** with the inverted mask surely makes a bit 0: the letter goes back to uppercase. **XOR** flips the bit: an uppercase letter becomes lowercase and a lowercase letter becomes uppercase. If XOR is done a second time with the same mask, the old value comes back; this property is also used in simple encryption methods.

A **bit shift** moves all the bits together. In a left shift by one, each bit moves up one position and its weight doubles; the value is multiplied by 2. A right shift divides a positive number by 2 and rounds down: 71 >> 1 = 35. For negative odd numbers `>>` and `/` give different results: `-3 >> 1` = −2 (rounds down), `-3 / 2` = −1 (truncates toward zero).

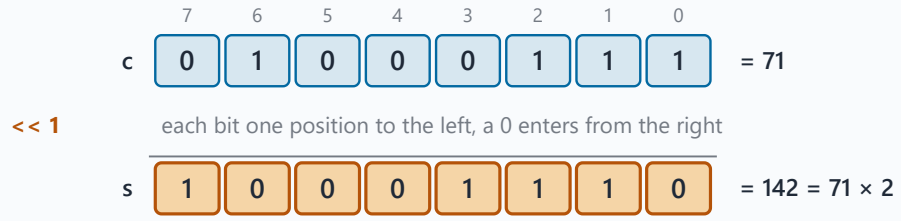


Figure 1.3.13: Shifting "G" = 71 one position to the left. Each bit moves up one position and a 0 enters from the right: 10001110 = 142 = 71 × 2.

In Java, bit operations are written with the operators `|` (OR), `&` (AND), `~` (NOT), `^` (XOR), `<<`, and `>>`. The `char` type in Java is a numeric code; so bit operations can be done directly with letters:

```
char c = 'G';
System.out.println((int) c);
System.out.println((char) (c | 32));
System.out.println((char) ('g' & ~32));
System.out.println((char) (c ^ 32));
System.out.println(c << 1);
System.out.println(c >> 1);
```

Output:

```
71
g
G
g
142
35
```

The result of a bit operation is an `int`. To see a letter on the screen, the result is converted back to a character with `(char)`. In the last two lines this conversion is not done, so numbers are printed. The same bits appearing as a letter in one place and as a number in another is one more example of the principle in 1.3.1.2, that the data type gives the meaning.

Common mistake

Always adding or subtracting 32 to change the case of a letter. "G" + 32 gives the right result, but "g" + 32 = 135 is an undefined code. Lowercasing with OR and uppercasing with AND do not change a letter that is already in the wanted state; this makes them safer. This method holds only for English letters in ASCII; the Turkish pairs "ı/I" and "i/İ" do not obey this rule.

1.3.4.3 Unicode and the code point

Unicode gives a single, universal number to the characters of the world's writing systems. This number is called the **code point** and is written in hexadecimal with the prefix "U+." The code point of "A" is U+0041; this is 65 in decimal and the same as its ASCII code. Unicode deliberately keeps its first 128 code points the same as ASCII. "ğ" is U+011F, "€" is U+20AC, and the grinning-face emoji is U+1F600. Today Unicode contains hundreds of thousands of characters, covering the Turkish letters, various alphabets, mathematical symbols, and emoji.

A code point is only a number. With which bytes the number is written in a file or in memory is a separate decision. The same code point can be stored with different Unicode transformation formats: UTF-8, UTF-16, and UTF-32 are the best known. If it is not known which format a text file was written in when it is opened, the characters look garbled.

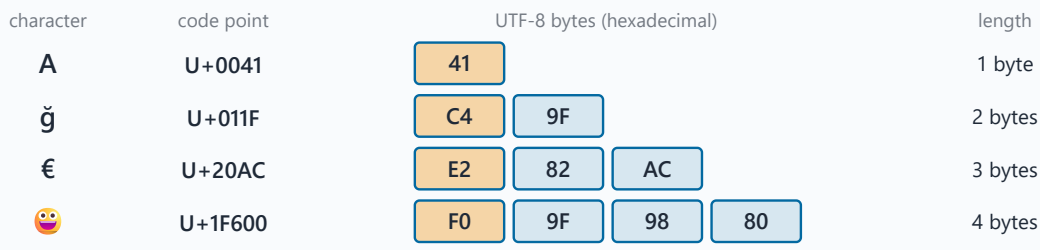


Figure 1.3.14: The code points and UTF-8 bytes of four characters. "A" (U+0041) is 1 byte: 41. "ğ" (U+011F) is 2 bytes: C4 9F. "€" (U+20AC) is 3 bytes: E2 82 AC. The grinning face (U+1F600) is 4 bytes: F0 9F 98 80.

1.3.4.4 UTF-8 encoding: "ğ"

UTF-8 writes Unicode code points as byte sequences of variable length. Small code points take few bytes and large ones many. Which byte has which role is told by the bits at the start of the byte. Table 1.3.9 summarizes the rules.

Code point range	Number of bits	Byte template	Example
U+0000 – U+007F	7	`0xxxxxxx`	"A" → 41
U+0080 – U+07FF	11	`110xxxxx 10xxxxxx`	"ğ" → C4 9F
U+0800 – U+FFFF	16	`1110xxxx 10xxxxxx 10xxxxxx`	"€" → E2 82 AC
U+10000 – U+10FFFF	21	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`	emoji → F0 9F 98 80

Table 1.3.9: UTF-8 templates. The bits of the code point are placed, from left to right, in the `x` positions.

The bits at the start of the templates are labels. A byte that begins with 0 is an ASCII character on its own. 110 means "the first byte of a two-byte character," 1110 "the first byte of a three-byte character," and 11110 "the first byte of a four-byte character." A byte that begins with 10 is a **continuation byte**. Thanks to these labels, even a program that starts reading in the middle of a text can find the character boundaries.

Example 1.3.8

Let us find the UTF-8 bytes of the letter "ğ."

1. **Code point:** Unicode assigns U+011F to "ğ." $011F_{16} = 1 \cdot 256 + 1 \cdot 16 + 15 = 287$.
2. **Binary form:** $287 = 256 + 16 + 8 + 4 + 2 + 1 = 100011111_2$. The number does not fit in 7 bits (it is not ASCII) but fits in 11 bits. So 2 bytes are needed. To fill out 11 bits, two 0s are added in front: 00100011111.
3. **Splitting:** The two-byte template is `110xxxxx 10xxxxxx` and has 5 + 6 = 11 free places. The 11 bits are split from the left as 5 + 6: 00100 | 011111.
4. **Placing:** The first byte is 110 + 00100 = 11000100 = $C4_{16}$. The second byte is 10 + 011111 = 10011111 = $9F_{16}$.
5. **Result:** "ğ" = C4 9F, 2 bytes.

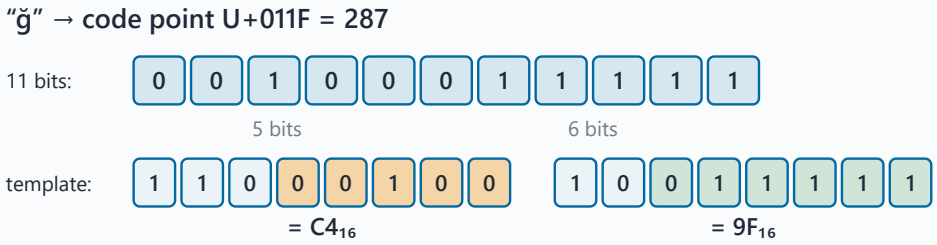


Figure 1.3.15: The 11 bits of "ğ" = U+011F = 287, placed as 5 + 6 into the template `110xxxxx 10xxxxxx`: 11000100 10011111 = C4 9F.

The same method works for "ç" too. The code point of "ç" is U+00E7 = 231; in 11 bits it is written 00011100111. The 5 + 6 split gives 00011 | 100111 and the bytes are 11000011 10100111 = C3 A7. All the letters specific to Turkish are in the range U+0080 – U+07FF; so in UTF-8 each takes 2 bytes. ASCII characters take 1 byte, "€" 3, and an emoji 4.

UTF-8's greatest advantage is its compatibility with ASCII. Because ASCII characters are written with a single byte and with their ASCII codes, old ASCII texts are also valid UTF-8. The great majority of web pages are written in UTF-8 today. If a UTF-8 file is mistakenly read with an old 8-bit table, the two bytes of "ğ" look like two separate characters; this is why meaningless letters such as "ÄŸ" appear on the screen.

1.3.4.5 Strings

A **string** is a data type made of ordered characters. A name, a sentence, or the whole content of a file is a string. The basic operations on strings are these:

- **Length:** the number of characters or code units.
- **Concatenation:** joining two strings end to end.
- **Search:** finding a character or a substring.
- **Comparison:** checking content equality and alphabetical order.

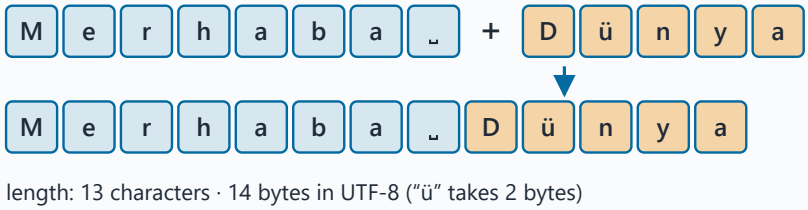


Figure 1.3.16: Concatenating “Merhaba ” and “Dünya” (Turkish for “Hello” and “world”). The result is a string of 13 characters; it takes 14 bytes in UTF-8, because “ü” is 2 bytes.

Figure 1.3.16 shows an important distinction: the number of characters seen on the screen and the number of bytes in memory are not always the same. “Merhaba Dünya” is 13 characters but takes 14 bytes in UTF-8. When allocating memory, calculating the size of a file, or choosing the length of a database field, one must know which of them is being counted. The unit of storage in memory is called a **code unit**: in UTF-8 a code unit is 1 byte, and in UTF-16, which Java uses, 2 bytes.

```
String s = "çağ";
System.out.println(s.length());
byte[] b = s.getBytes(java.nio.charset.StandardCharsets.UTF_8);
System.out.println(b.length);
for (byte x : b) {
    System.out.printf("%02X ", x);
}
System.out.println();
```

Output:

```
3
5
C3 A7 61 C4 9F
```

In Java the `length()` method counts UTF-16 code units; for “çağ” it gives 3. `getBytes` turns the string into UTF-8 bytes: “ç” C3 A7, “a” 61, “ğ” C4 9F. The total is 5 bytes. Because an emoji takes two code units in UTF-16, its `length()` comes out as 2; the idea of a “number of characters” is subtler in Unicode than it looks.

Common mistake

Taking the number of characters to be the number of bytes. In UTF-8 “çağ” (Turkish for “era”) is 3 characters and 5 bytes (ç 2 + a 1 + ğ 2). Ten Turkish letters may not fit in a 10-byte field.

1.3.5 Structured information and common mistakes

1.3.5.1 From bits to files

Bits do not carry meaning by themselves; they gain meaning by being arranged in larger structures. These structures are built layer by layer.

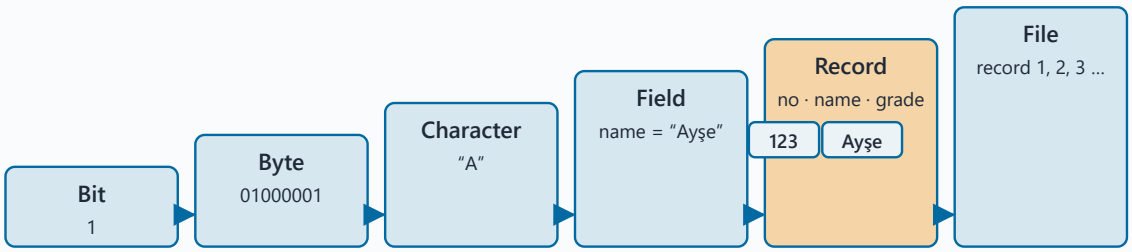


Figure 1.3.17: Bit, byte, character, field, record, and file. Each level is built from the one before: eight bits make a byte, one or a few bytes make a character, characters make a field, fields make a record, and records make a file.

- **Bit and byte:** the smallest units. Eight bits make a byte.
- **Character:** one byte in ASCII, one to four bytes in UTF-8.
- **Field:** a single meaningful piece of information; for example a name (“Ayşe”) or a grade (85).
- **Record:** a collection of related fields. A student record may consist of the fields number, name, and grade.
- **File:** a sequence of records and other data stored on disk.

How the bytes inside a file are to be read is defined by the **file format**: the order of the fields, how many bytes each field has, and in which encoding it is written. The same bytes are a student list to one program and meaningless numbers to another. This is why a jumble of characters appears when an image file is opened with a text editor: the bytes are interpreted with the wrong format.

1.3.5.2 Common mistakes in data representation

The mistakes scattered through the sections on data representation are collected in Table 1.3.10 for quick reference.

Common mistake	The correct way
Reading a number without its base: “1101 is one thousand one hundred one”	1101_2 = 13. The base is indicated with a subscript or the `0b` / `0x` prefix.
Taking 10_{16} to be ten	10_{16} = 16; in hexadecimal, ten is written with A.
Only flipping the bits in two’s complement	11111010 is the one’s complement (not −5); adding 1 gives 11111011 = −5.
Comparing floating-point results with `==`	0.1 + 0.2 ≠ 0.3; check whether |a − b| is smaller than a small tolerance.
Confusing a digit character with a number	The code of the character “7” is 55; its numeric value is “7” − “0” = 7.
Taking the number of characters to be the number of bytes	In UTF-8 “çağ” is 3 characters and 5 bytes (ç 2 + a 1 + ğ 2).
Confusing carry with signed overflow	Carry is a problem of the unsigned interpretation, overflow of the signed one; one can occur without the other.

Table 1.3.10: Common mistakes in data representation and their corrections.

1.3.6 Logic gates and adders

In the earlier sections, addition was always done on paper. The processor, however, does addition with electronic circuits. The building blocks of these circuits are logic gates. This section shows how a few gates come together to add two bits, and then multibit numbers.

1.3.6.1 Logic gates and truth tables

Definition 1.3.6

A **logic gate** is a basic circuit that produces a one-bit output from a one- or two-bit input. The behavior of a gate is fully defined by its **truth table**, which lists the output for every combination of inputs.

There are four basic gates:

- **AND** gives 1 if both inputs are 1.
- **OR** gives 1 if at least one of the inputs is 1.
- **XOR** (exclusive OR) gives 1 if the inputs differ.
- **NOT** reverses its single input.

A	B	AND	OR	XOR	NOT A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0

A	B	AND	OR	XOR	NOT A
1	1	1	1	0	0

Table 1.3.11: Truth table of the four basic gates.

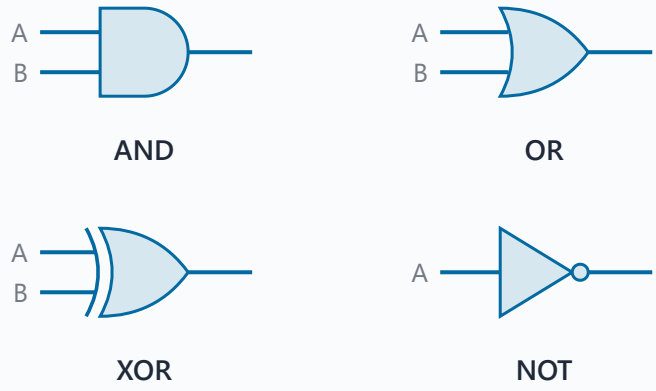


Figure 1.3.18: The circuit symbols of the AND, OR, XOR, and NOT gates. The inputs are on the left and the output on the right. The small circle at the tip of NOT means "inversion."

The only difference between OR and XOR is in the last row: when both inputs are 1, OR gives 1 and XOR gives 0. This behavior of XOR produces the result digit in binary addition, because the digit of 1 + 1 is 0. AND gives the carry produced in 1 + 1.

Gates are built from a few transistors. A transistor is a small electronic switch opened and closed by a voltage. A processor today contains billions of transistors, and all the operations in the processor are made up of combinations of these basic gates. By connecting gates to one another, more complex logical expressions are computed. For example, in the expression *X = NOT (A AND B) OR C*, first the parentheses are evaluated, then NOT, and OR last. For A = 1, B = 1, C = 0: A AND B = 1, NOT 1 = 0, and 0 OR 0 = 0. If C = 1, a single 1 would be enough for the OR, so the result would be 1 whatever A and B are.

1.3.6.2 The half adder

The sum of two one-bit numbers can be at most $1 + 1 = 2 = 10_2$; the result is written in two bits. The right bit is the result digit (SUM) and the left bit is the carry (CARRY). Table 1.3.12 lists the four cases.

A	B	A + B	CARRY	SUM
0	0	$0 = 00_2$	0	0
0	1	$1 = 01_2$	0	1
1	0	$1 = 01_2$	0	1
1	1	$2 = 10_2$	1	0

Table 1.3.12: Truth table of the half adder. The SUM

column is the same as the table of XOR and the CARRY column the same as the table of AND.

Looking carefully at the table, one sees that the SUM column is the same as the table of XOR and the CARRY column the same as that of AND. So two gates are enough to add two bits.

Definition 1.3.7

A **half adder** is a circuit that adds two one-bit inputs (A, B): *SUM = A XOR B* gives the result digit and *CARRY = A AND B* gives the carry.

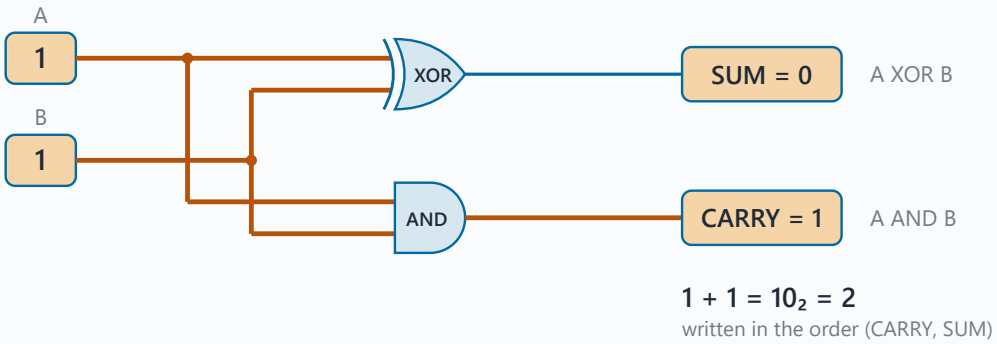


Figure 1.3.19: The half adder in the case 1 + 1. A and B go to both gates. XOR(1, 1) = 0 gives the result digit and AND(1, 1) = 1 gives the carry: (CARRY, SUM) = 10_2 = 2.

Figure 1.3.19 shows the circuit's most interesting case. For 1 + 1, XOR gives 0 and AND gives 1. The result, in the order (CARRY, SUM), is 10_2 = 2. Like 9 + 1 = 10 in the decimal system, the position overflows and the carry goes to the next position up. The half adder's name comes from here: it has no input to take an incoming carry, and does only half of the job of addition.

1.3.6.3 The full adder

When multibit numbers are added, every column except the rightmost can receive a carry from the column to its right. So three bits are added in each column: A, B, and the incoming carry (Cin). The sum of three bits is at most 3 = 11_2; it is again written in two bits.

Definition 1.3.8

A **full adder** is a circuit that adds three one-bit inputs (A, B, Cin) and produces the result digit (SUM) and the outgoing carry (Cout). It is built from two half adders and one OR gate:

> t = A XOR B, SUM = t XOR Cin, Cout = (A AND B) OR (t AND Cin)

The first half adder adds A and B and produces the intermediate result *t*. The second half adder adds the incoming carry to *t*. A carry can come out of one of the two half adders; it cannot come out of both. The OR gate combines these two possible carries.

Example 1.3.9

Let us find the outputs of the full adder for A = 1, B = 1, Cin = 1.

1. **First half adder:** t = 1 XOR 1 = 0; first carry 1 AND 1 = 1.
2. **Second half adder:** SUM = 0 XOR 1 = 1; second carry 0 AND 1 = 0.
3. **OR:** Cout = 1 OR 0 = 1.
4. **Check:** (Cout, SUM) = 11_2 = 3 = 1 + 1 + 1.

1.3.6.4 A four-bit adder: 0110 + 0111

Four full adders are placed side by side and the carry output of each is connected to the carry input of the one to its left. This gives a circuit that adds four-bit numbers. The incoming carry of the rightmost adder is 0. The pseudocode in the slides describes the work the circuit does from right to left:

```
a ← 0110  ; 6
b ← 0111  ; 7
c ← 0     ; initial carry
for i = 0, 1, 2, 3: ; right to left
  t ← a[i] XOR b[i]
  s[i] ← t XOR c
  c ← (a[i] AND b[i]) OR (t AND c)
print(c, s)
```

i	a[i]	b[i]	incoming c	t	s[i]	outgoing c
0	0	1	0	1	1	0
1	1	1	0	0	0	1
2	1	1	1	0	1	1

i	a[i]	b[i]	incoming c	t	s[i]	outgoing c
3	0	0	1	0	1	0

Table 1.3.13: Trace of the full-adder chain for 0110 + 0111 (6 + 7). In each row `c` is the incoming carry and the last column is the outgoing carry.

The table is read row by row as follows. At $i = 0$, 0 + 1 + 0 is added: $t = 1$, $s_0 = 1$; since the two inputs are not both 1, the carry is 0. At $i = 1$, 1 + 1 + 0 = 2 = 10_2: digit 0, carry 1. This is the binary counterpart of 5 + 5 = 10 in decimal. At $i = 2$ the incoming carry is added too: 1 + 1 + 1 = 3 = 11_2, digit 1, carry again 1. At $i = 3$, 0 + 0 + 1 = 1; digit 1, carry 0. Because the last carry is 0, the result fits in 4 bits.

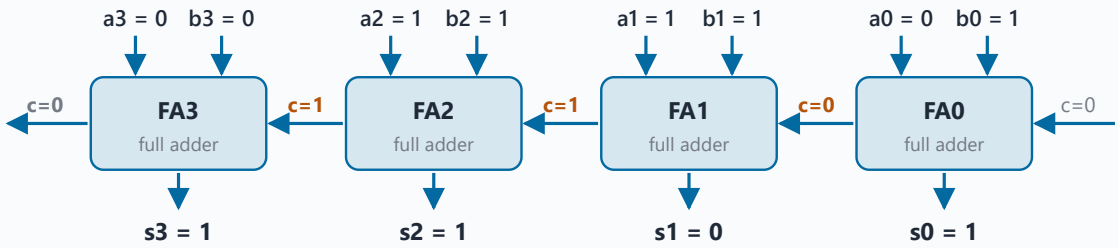


Figure 1.3.20: A chain of four full adders (FA0–FA3), for 0110 + 0111. The carry moves from right to left: 0 comes out of FA0, 1 out of FA1, 1 out of FA2, and 0 out of FA3. The result is 1101_2 = 13 = 6 + 7.

The result is 1101_2 = 13 = 6 + 7. In this arrangement the carry moves from right to left like a wave; the name of the circuit comes from here: **ripple-carry adder**. Each full adder has to wait for the carry of the one to its right. As the number of bits grows, the waiting time, that is, the delay, grows too: in a 64-bit ripple-carry adder the leftmost position waits for 63 adders to finish in turn. Real processors therefore use more complex adders that compute the carry in advance; but all of them are built from the same gates.

This circuit is directly connected to the concepts in 1.3.2. The carry coming out of the last full adder is the carry bit in 1.3.2.9. For example, if 1011 + 0111 (11 + 7) is added, a carry of 1 comes out in every column; what remains in 4 bits is 0010 = 2, and the last carry 1 shows that the result does not fit in 4 bits. Thanks to two's complement, the same circuit also subtracts: for $a - b$ the bits of b are flipped with NOT gates and the rightmost incoming carry is made 1. In this way the rule "flip the bits, add 1" is implemented in hardware with a single circuit.

Common mistake

Trying to build multibit addition from half adders. The half adder has no incoming-carry input; it can be used only in the rightmost column. In the other columns three bits are added, so a full adder is needed.

1.3.7 The stored program and the CPU

An adder only adds. What makes a computer general-purpose is the existence of a program that says which operation is to be done in which order. This section explains how a program sits in memory and how the processor runs it step by step.

1.3.7.1 The stored-program concept

Some of the first electronic calculators required the wires to be reconnected by hand for a new calculation. Changing a program could take days. An idea put forward in the mid-1940s changed this radically: instructions, too, could be placed in the same memory as the data, as numbers.

Definition 1.3.9

In the **stored-program** approach, the program's instructions and the data to be processed are stored in the same memory as numeric codes. The processor reads the instructions from memory in order and executes them.

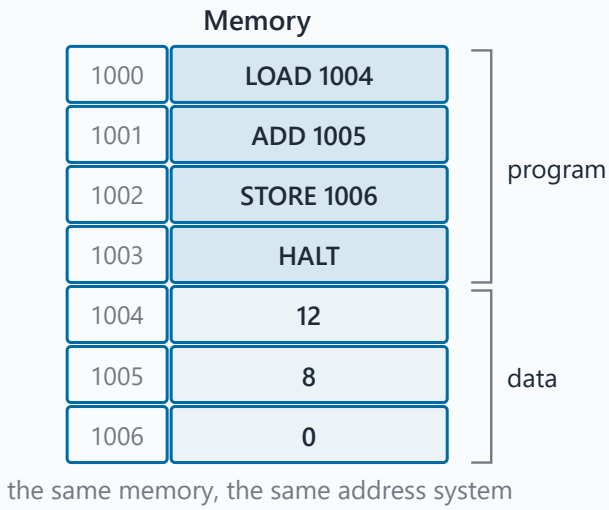


Figure 1.3.21: Program and data in the same memory. At addresses 1000–1003 sit the instructions (LOAD 1004, ADD 1005, STORE 1006, HALT), at addresses 1004–1006 the data (12, 8, 0). Both use the same address system.

This approach has three important consequences.

- **Flexibility:** The same hardware does different tasks with different programs. On a computer, a spreadsheet, a game, and a browser run on the same processor.
- **Updating:** Behavior can be changed with software without changing the hardware. Updating a phone is writing new instructions into memory.

- **General purpose:** A single system can carry out computing, communication, and media tasks together. A program can even read, produce, or translate another program as data; compilers do exactly this.

1.3.7.2 The fetch–decode–execute cycle

The processor executes the program in memory with a three-stage cycle that repeats without stopping. Before describing this cycle, two registers need to be introduced.

- The **program counter** (PC) holds the memory address of the next instruction.
- The **instruction register** (IR) holds the instruction fetched from memory and being executed at that moment.

Definition 1.3.10

The **fetch–decode–execute cycle** consists of three stages:

1. **Fetch:** the instruction at the address in the PC is read and placed in the IR; the PC is incremented by one.
2. **Decode:** The control unit splits the instruction in the IR: the *opcode* says what to do, the *operand* says what to do it with.
3. **Execute:** The ALU or another unit carries out the operation; the result is written to a register or to memory.

Then the cycle starts again with the fetch stage.

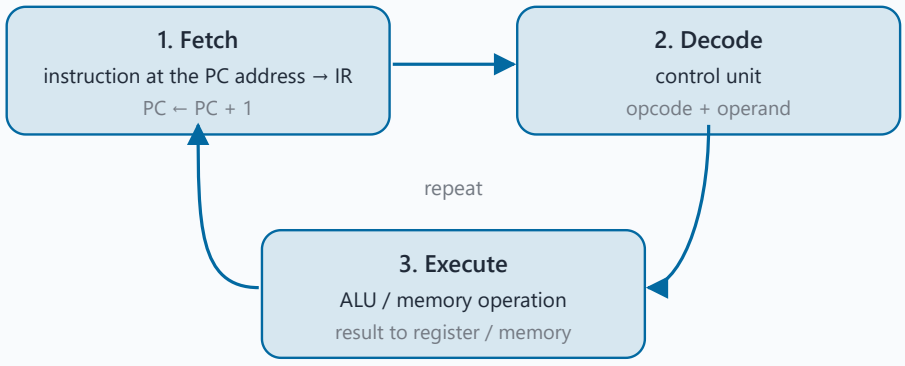


Figure 1.3.22: The fetch, decode, and execute stages form a cycle. In fetch, the instruction at the PC address comes into the IR and the PC goes up by one. In decode, the control unit splits the opcode and the operand. In execute, the ALU or memory operation is done and the result is written to a register or to memory.

There is a reason the PC is incremented in the fetch stage, that is, before the instruction is even executed. The great majority of instructions require moving on to the next instruction; incrementing the PC in advance does this automatically. Branch instructions, on the other hand, change this order in the execute stage by overwriting the PC with a new address (1.3.7.8).

Although the cycle is described as reading from memory, instructions and data often come not from RAM but from the **cache** inside the processor. Registers and the cache are much faster than RAM. This detail does not change the logic of the cycle; it only raises its speed.

1.3.7.3 The main parts of the CPU

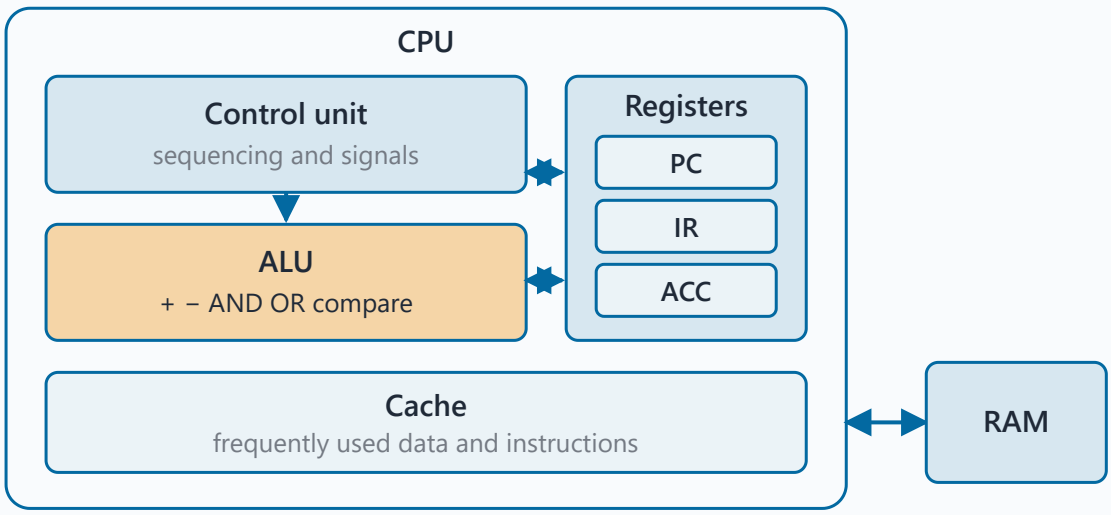


Figure 1.3.23: Inside the CPU. The control unit manages the sequence and the signals; the ALU does addition, subtraction, AND, OR, and comparison; the registers (PC, IR, ACC) hold intermediate values; the cache brings frequently used data and instructions from RAM and keeps them nearby.

Figure 1.3.23 shows the four main parts of a processor.

- The **control unit** manages the order of instructions and the movement of data. It performs the decode stage and sends signals that tell the other units what to do and when.
- The **arithmetic logic unit** (ALU) performs arithmetic and logical operations. The adder in 1.3.6.4 is a part of the ALU.
- **Registers:** small, fast storage cells used during processing. Besides the PC and IR, the example processor in these notes has an **accumulator** (ACC) in which calculation results build up.
- **Cache:** keeps frequently used data and instructions close to the processor.

1.3.7.4 Buses: reading from memory

The CPU is connected to memory and the I/O devices by bundles of parallel wires called **buses**. There are three separate buses, and each has a different job.

Bus	Job	Direction
Address bus	Specifies the memory or device location to be operated on.	one-way: leaves the CPU
Control bus	Carries read, write, and timing signals.	mostly leaves the CPU
Data bus	Carries data among the CPU, memory, and devices.	two-way

Table 1.3.14: The three buses that connect the CPU to memory and devices.

Example 1.3.10

The CPU is about to read the value at address 1004; the cell holds 12. The three buses are used in turn as follows:

1. **Address bus:** The CPU sends which cell it wants: 1004.
2. **Control bus:** The CPU sends the READ signal. For a write, the WRITE signal would go.
3. **Data bus:** Memory sends the cell's contents, 12, to the CPU. The value is loaded into the ACC.

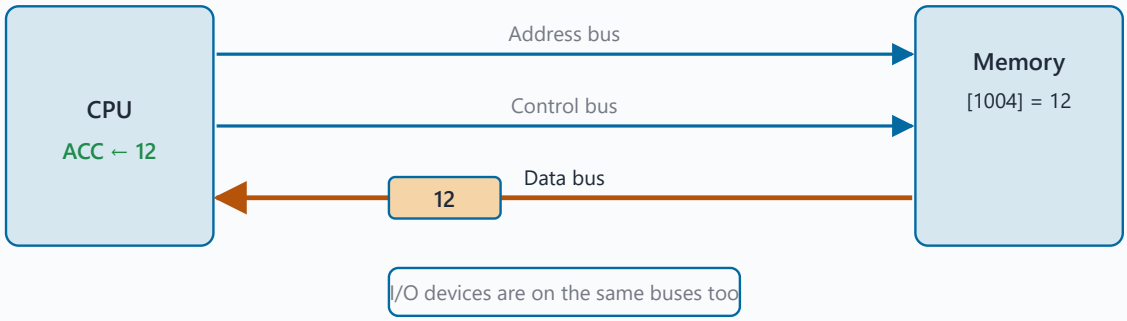


Figure 1.3.24: The last step of reading from memory. 1004 has gone out on the address bus and the READ signal on the control bus; memory sends the cell's contents, 12, to the CPU over the data bus.

In a write, the address and control buses are again used from the CPU toward memory, but this time data flows from the CPU to memory. This is why the data bus is two-way. The width of the address bus determines the number of addresses that can be specified: a 16-bit address bus can specify 2^{16} = 65,536 different addresses.

Common mistake

Thinking that the address also travels on the data bus. The number “1004” goes to memory on the address bus; the data bus carries only the cell’s contents (12). The control bus says what is to be done (READ or WRITE).

1.3.7.5 Memory addresses

Memory can be pictured as a long array of numbered cells. Each cell has an **address**, and instructions reach data with this address. The small example memory used in these notes is shown in Table 1.3.15.

Address	Contents	Meaning
1000	`LOAD 1004`	ACC ← mem[1004]
1001	`ADD 1005`	ACC ← ACC + mem[1005]
1002	`STORE 1006`	mem[1006] ← ACC
1003	`HALT`	stop
1004	12	data
1005	8	data
1006	0	result location

Table 1.3.15: Example memory contents. The first four cells are instructions and the last three cells are data.

The notation “mem[1004]” in the table means “the contents of the cell at address 1004.” Address and contents are different things: at address 1004 there is 12. It is like the difference between a house address and the person living in that house.

In memory there is no physical distinction between instructions and data. Instructions also sit in memory as numbers; the same bit sequence can be interpreted as an instruction or as data. Which one a cell is depends on whether the PC reaches that cell. This is why the HALT instruction is necessary. Without HALT the CPU would move on from 1003 to 1004 and take the 12 there for an instruction and try to run it.

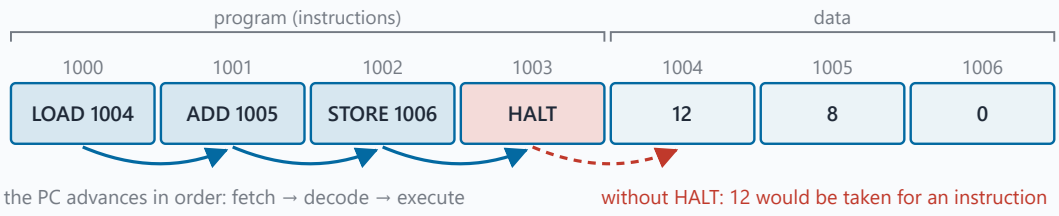


Figure 1.3.25: The PC starts at 1000 and fetches the instructions at 1001, 1002, and 1003 in order. The HALT at 1003 stops the CPU; without HALT the 12 at 1004 would be taken for an instruction.

1.3.7.6 Multibyte values and byte order

In real computers memory is **byte-addressed**: each address holds 1 byte. A 32-bit `int` is 4 bytes and spreads across 4 consecutive addresses. A question then arises: which of the four bytes is written to the lowest address? This order is called **endianness**, or byte order.

In the value 0x12345678 the most significant byte is 12 and the least significant byte is 78, just as in the number 1234 the 1 is in the thousands position and the 4 in the ones position. There are two common arrangements.

- **Big-endian:** the most significant byte is written to the lowest address. From address 100 to 103, 12 34 56 78 is read; the number is laid out in the order it is written on paper. Network protocols use this order; so it is also called **network byte order**.
- **Little-endian:** the least significant byte is written to the lowest address. From address 100 to 103, 78 56 34 12 is read. x86-64 and common ARM systems work in this order.

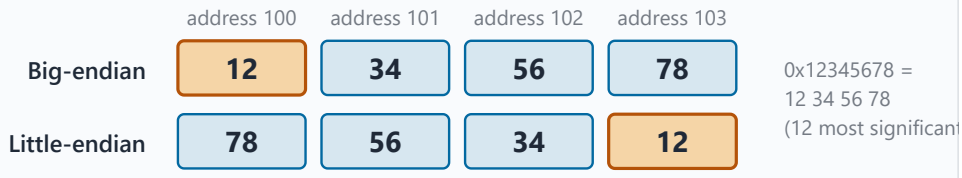


Figure 1.3.26: How 0x12345678 is laid out at addresses 100–103. In the big-endian arrangement the most significant byte (12) is at address 100; in the little-endian arrangement it is at address 103.

Both arrangements work correctly; trouble arises only when raw binary data is carried between two systems. The bytes 78 56 34 12 written by a little-endian machine, if read in one go on a big-endian machine, are seen as 0x78563412. File formats and network protocols therefore state the byte order explicitly. In Java the order can be chosen with the `ByteBuffer` class:

```
java.nio.ByteBuffer bb = java.nio.ByteBuffer.allocate(4);
bb.order(java.nio.ByteOrder.LITTLE_ENDIAN);
bb.putInt(0x12345678);
for (byte x : bb.array()) {
    System.out.printf("%02X ", x);
}
System.out.println();
```

Output:

```
78 56 34 12
```

For one-byte values, for example an ASCII character, there is no order problem. UTF-8 is also unaffected by byte order because it is read byte by byte; “ğ” is written in the order C4 9F on every machine.

1.3.7.7 Running a program step by step

The parts introduced so far come together in the execution of the small program in Table 1.3.15. The program adds two numbers and writes the result to memory. To the right of each line, after the semicolon, the meaning of the instruction is written:

```
1000: LOAD  1004 ; ACC ← mem[1004]
1001: ADD   1005 ; ACC ← ACC + mem[1005]
1002: STORE 1006 ; mem[1006] ← ACC
1003: HALT       ; stop
1004: 12         ; data
1005: 8          ; data
1006: 0          ; result
```

In this example processor, calculations are done in a single register, the ACC. The instructions mean: `LOAD a` loads the contents of memory address *a* into the ACC; `ADD a` adds those contents to the ACC; `STORE a` writes the ACC to address *a*; `HALT` stops the processor. At the start PC = 1000, the address of the first instruction, and ACC = 0.

Step	Stage	What happens?	PC	IR	ACC	mem[1006]
0	start	—	1000	—	0	0
1	fetch	1000 goes onto the address bus, LOAD 1004 arrives in the IR over the data bus	1001	LOAD 1004	0	0
2	decode	opcode LOAD, address 1004	1001	LOAD 1004	0	0
3	execute	1004 is read, 12 is loaded into the ACC	1001	LOAD 1004	12	0
4	fetch	ADD 1005 arrives in the IR	1002	ADD 1005	12	0
5	decode	opcode ADD, address 1005	1002	ADD 1005	12	0
6	execute	the 8 at 1005 is read, the ALU computes 12 + 8 = 20	1002	ADD 1005	20	0
7	fetch	STORE 1006 arrives in the IR	1003	STORE 1006	20	0
8	decode	opcode STORE, address 1006	1003	STORE 1006	20	0

Step	Stage	What happens?	PC	IR	ACC	mem[1006]
9	execute	the 20 in the ACC is written to 1006 over the data bus	1003	STORE 1006	20	20
10	fetch	HALT arrives in the IR	1004	HALT	20	20
11	decode	opcode HALT → the CPU stops	1004	HALT	20	20

Table 1.3.16: Fetch–decode–execute trace of the program. Each row shows the register and memory state after the step has finished.

Several regular patterns stand out in the trace. The PC changes only in the fetch stages and goes up by one each time. The IR takes the new instruction at each fetch and does not change until that instruction is done. The ACC changes only in the execute stages of LOAD and ADD; memory changes only in the execution of STORE. When the program ends, mem[1006] = 20. Four instructions have been executed in four fetch–decode–execute rounds; because HALT has no work to execute, its round ends in the decode stage.

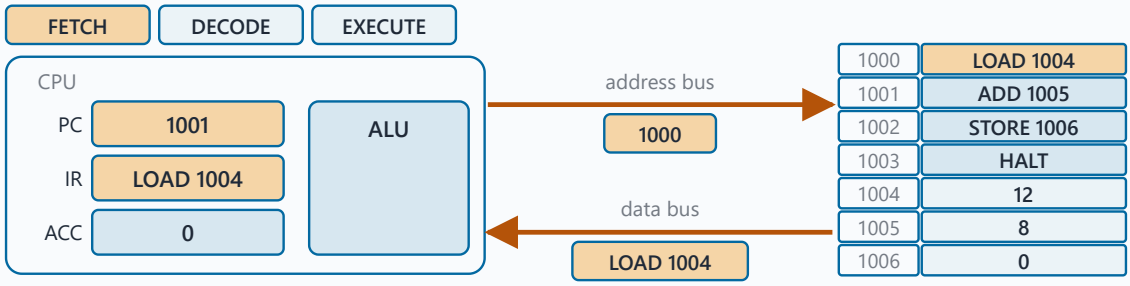


Figure 1.3.27: The first fetch stage (step 1). The 1000 in the PC goes onto the address bus, the instruction LOAD 1004 arrives in the IR over the data bus, and the PC goes up to 1001.

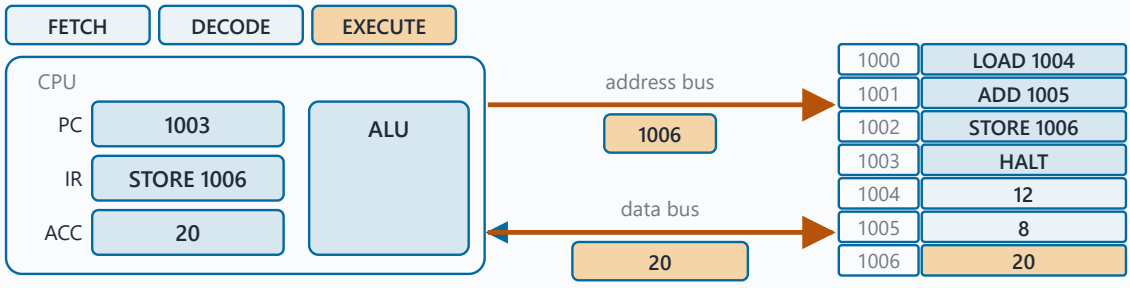


Figure 1.3.28: The execute stage of STORE (step 9). The 20 in the ACC is written to address 1006 over the data bus.

Example 1.3.11

What is the result if the places of STORE and ADD in the program are swapped (1001: STORE 1006, 1002: ADD 1005)?

1. **LOAD 1004:** ACC = 12.
2. **STORE 1006:** the current ACC is written; mem[1006] = 12.
3. **ADD 1005:** ACC = 12 + 8 = 20; but this value is never written to memory.
4. **HALT:** it stops. Result: ACC = 20, mem[1006] = 12; again 4 instructions have been fetched.

The CPU executes instructions in the order they are written; when the order changes, the result changes.

1.3.7.8 Branching: how does the PC change in a loop?

The previous program executed the instructions once from start to end. Real programs make decisions and repeat work. For this, **branch** instructions are used. A branch instruction overwrites the PC with a new address in the execute stage; so the next fetch happens from somewhere else. The example processor has two branch instructions:

- `JUMP a`: unconditional branch; PC ← *a*.
- `JUMPZ a`: conditional branch; if ACC = 0, PC ← *a*; otherwise the PC stays as it is and the next instruction follows.

The program below counts `count` down from 2 and adds 2 + 1 to `sum`. Addresses 1010, 1011, and 1012 hold `count`, `sum`, and the constant 1, respectively:

```
1000: LOAD  1010  ; ACC ← count
1001: JUMPZ 1008  ; jump if ACC = 0
1002: ADD   1011  ; ACC ← ACC + sum
1003: STORE 1011  ; sum ← ACC
1004: LOAD  1010
1005: SUB   1012  ; ACC ← ACC – 1
1006: STORE 1010  ; count ← ACC
1007: JUMP  1000  ; PC ← 1000
1008: HALT
1010: 2   1011: 0   1012: 1
```

The program is the machine-level counterpart of this loop in Java: `while (count != 0) { sum = sum + count; count = count - 1; }`. Lines 1000–1001 are the loop condition, lines 1002–1006 the loop body, and line 1007 the return to the start.

Instr. no.	Executed	PC	ACC	count	sum	Explanation
—	start	1000	0	2	0	
1	1000: LOAD 1010	1001	2	2	0	ACC ← count
2	1001: JUMPZ 1008	1002	2	2	0	ACC ≠ 0, no jump
3–4	ADD 1011, STORE 1011	1004	2	2	2	sum = 2 + 0
5–7	LOAD, SUB, STORE 1010	1007	1	1	2	count = 2 − 1
8	1007: JUMP 1000	1000	1	1	2	the PC's 1008 is replaced by 1000
9–16	second round (the same 8 instructions)	1000	0	0	3	sum = 1 + 2, count = 0
17	1000: LOAD 1010	1001	0	0	3	ACC ← 0
18	1001: JUMPZ 1008	1008	0	0	3	ACC = 0, jump taken
19	1008: HALT	1009	0	0	3	the CPU stops

Table 1.3.17: Trace of the branching program. Each row shows the state after the instruction has been executed. The PC column is the address to be used at the next fetch.

The critical moments of the trace are the two branch instructions. At instruction 8, `JUMP 1000` is executed. In the fetch stage the PC had already gone up to 1008; execution writes 1000 over this value and the loop goes back to the start. At instruction 18, `JUMPZ` this time jumps, because the count loaded into the ACC is 0. The PC becomes 1008 and the loop body is skipped. In all, 19 instructions have been fetched: 8 + 8 for two full rounds and 3 for the exit (LOAD, JUMPZ, HALT).

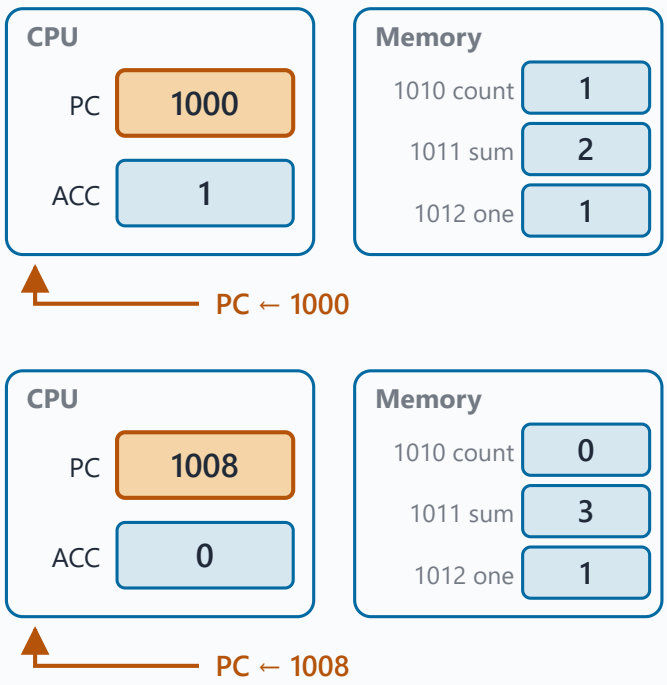


Figure 1.3.29: Two moments of branching. (a) At the end of the first round, `JUMP 1000` : the PC becomes 1000, count is 1, and sum is 2. (b) In the last round `JUMPZ` jumps: because ACC = 0, the PC moves to 1008.

The general formula for the number of instructions can also be drawn from the trace. If the count starts from k, the loop turns k rounds and each round is 8 instructions; 3 more instructions are executed at the exit. The total is $8k + 3$ instructions: 19 for $k = 2$ and 43 for $k = 5$. The final value of the `sum` variable is $k + (k - 1) + \ldots + 1$; 15 for $k = 5$. This count is the first example of how a program's running time grows with the size of its input.

Common mistake

Thinking that a branch instruction changes the PC in the fetch stage. In fetch, the PC goes up by one as for every instruction (1007 → 1008). The jump is made in the execute stage by overwriting this new value. In a conditional branch, if the condition does not hold the PC is left alone and the incremented value stays in force.

1.3.8 Computer architecture

How the parts of a computer are arranged and how the instructions the programmer sees are designed is called **architecture**. This section covers the arrangement underlying today's computers, an alternative to it, how instructions are written in bits, and the factors that determine performance.

1.3.8.1 The von Neumann architecture

Today's general-purpose computers rest on an arrangement that uses the stored-program idea. This arrangement is called the **von Neumann architecture**, after a report John von Neumann wrote in 1945.

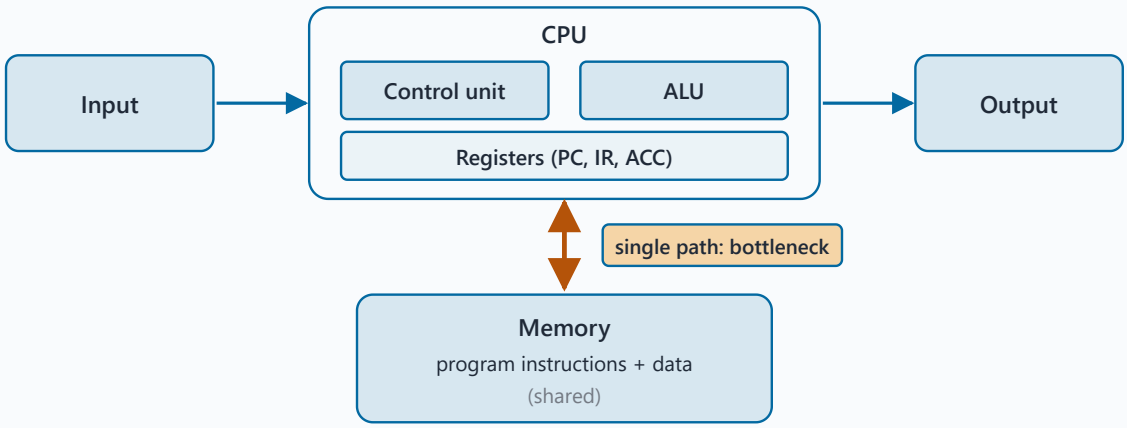


Figure 1.3.30: The von Neumann architecture. The CPU (control unit, ALU, registers) is connected by a single path to the shared memory that holds program instructions and data together. Input and output devices connect to the CPU.

The architecture has three defining properties.

- Program instructions and data are kept in **shared memory**.
- The CPU processes instructions **in sequence**; branch instructions can change this order.
- Instructions and data come **over the same path**.

The last property has a cost. While the CPU is fetching an instruction, it cannot read data over the same path; the two are carried one after the other. Moreover, over the years processors have developed much faster than memory. As a result, CPU–memory transfer limits performance. This situation is called the **von Neumann bottleneck**. The cache exists to ease this bottleneck.

1.3.8.2 Von Neumann and Harvard

A different way around the bottleneck is to keep instructions and data in separate memories. This arrangement is called the **Harvard architecture**; its name comes from the Harvard Mark I computer, which was built this way.



Figure 1.3.31: On the left, von Neumann: a single memory and a single path. On the right, Harvard: separate instruction and data memories and separate paths. The CPU only reads from the instruction memory; it reads from and writes to the data memory.

Property	von Neumann	Harvard
Memory	Instructions and data in shared memory	Instructions and data in separate memories
Bus	One path: instructions and data are carried in turn	Two paths: instructions and data are read at the same time
Flexibility	A program can be loaded and modified like data	Program memory is mostly fixed (flash)
Common use	General-purpose computers	Microcontrollers, signal processors

Table 1.3.18: Comparison of the von Neumann and Harvard architectures.

The advantage of the Harvard arrangement is speed: while one instruction is being fetched, the data of the previous instruction can be read at the same time. The price is flexibility. In a microcontroller the program is written to flash memory once and does not change while the device runs; for a washing machine this is enough. A general-purpose computer, however, has to load new programs every day.

Today's processors combine the two arrangements. Main memory is shared; programs and data are loaded into the same RAM. The first-level (L1) cache closest to the core, however, is separate for instructions and data. This hybrid arrangement is called the **modified Harvard** architecture. The programmer sees the shared memory, while the processor makes use of two separate paths inside.

1.3.8.3 Encoding instructions

As a requirement of the stored-program idea, instructions are written in bits as well. A machine instruction usually carries two pieces of information: the operation to be done and the value or address to be used.

- **Opcode** (operation code): defines the type of operation; for example ADD.
- **Operand:** the register, value, or address to be used in the operation; for example 1005 or R1, R2.
- **Instruction format:** determines the number and position of the bit fields. Every architecture has its own format.
- **Instruction set:** the complete set of instructions the processor can understand directly.

A hypothetical format is used for the example processor in these notes: the instruction is 16 bits, the first 4 bits are the opcode, and the remaining 12 bits are the address. The opcodes are HALT = 0000, LOAD = 0001, ADD = 0010, STORE = 0011.

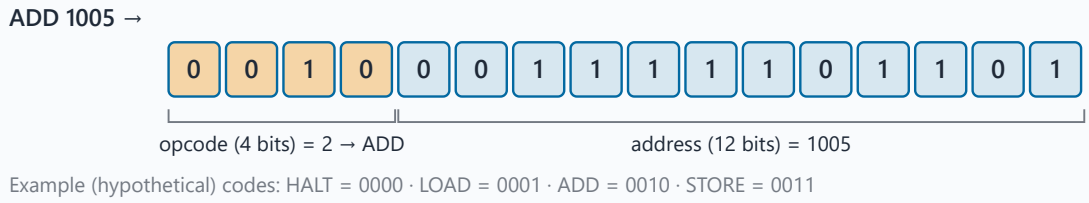


Figure 1.3.32: A 16-bit instruction word. For ADD 1005 the first 4 bits are the opcode 0010 (ADD), and the remaining 12 bits are the address 1005 = 001111101101.

Example 1.3.12

Let us encode the four instructions of the program in Table 1.3.15 with this format.

1. **Addresses:** 1004 = $3EC_{16}$ = 0011 1110 1100; 1005 = $3ED_{16}$ = 0011 1110 1101; 1006 = $3EE_{16}$ = 0011 1110 1110. (Check: $3 \cdot 256 + 14 \cdot 16 + 12 = 1004$.)
2. **LOAD 1004:** 0001 0011 1110 1100 = $13EC_{16}$.
3. **ADD 1005:** 0010 0011 1110 1101 = $23ED_{16}$.
4. **STORE 1006:** 0011 0011 1110 1110 = $33EE_{16}$.
5. **HALT:** 0000 0000 0000 0000 = 0000_{16}; the address field is not used.

> With a 12-bit address field, 2^{12} = 4,096 different addresses (0–4095) can be specified. For a larger memory the address field must be widened.

This encoding makes the observation in 1.3.7.5 concrete. The number 12 at address 1004 is written in 16 bits as 0000 0000 0000 1100. The first four bits of this sequence are 0000, that is, the opcode of HALT. If the same bit sequence is read as data it is 12; if read as an instruction it is a HALT instruction.

1.3.8.4 Instruction set design: RISC and CISC

There are two main approaches to how an instruction set is designed.

Property	RISC	CISC
Instruction set	A small number of simple instructions	A large number of complex instructions
Instruction length	Fixed (e.g., 32 bits)	Variable (1–15 bytes in x86-64)
Memory access	Only LOAD / STORE instructions	Operation instructions can take a memory operand directly
Example	ARM, RISC-V	x86-64

Table 1.3.19: Comparison of the RISC and CISC approaches.

In the **RISC** (reduced instruction set computer) approach, instructions are simple and of equal length. Only LOAD and STORE access memory; calculations are done only on registers. In the **CISC** (complex instruction set computer) approach, a single instruction can read from memory, calculate, and write to memory. The difference shows on the operation `a = a + b` in memory.

RISC:

```
LOAD  R1, a
LOAD  R2, b
ADD   R1, R1, R2
STORE R1, a
```

CISC (x86-64):

```
MOV EAX, [b]
ADD [a], EAX
```

RISC needs four instructions: two reads, one addition, one write. In x86-64 the job is finished with two instructions: the first brings *b* into the EAX register, and the second adds it directly to *a* in memory. The square brackets mean “the memory contents at this address.” But fewer instructions does not mean less work. The second instruction is again split inside the processor into read, add,

and write steps. Today x86-64 processors also execute complex instructions by turning them into small RISC-like steps internally. The great majority of phones use ARM and most personal computers use x86-64; in recent years ARM-based laptops have become common as well.

1.3.8.5 Architectural limits

However fast a computer is, it works within some limits. Most of these limits are general names for phenomena met separately in these notes.

- **Finite memory:** The amount of data that can be stored at one time is limited.
- **Finite speed:** Every instruction consumes time and energy.
- **Representation errors:** A limited number of bits can produce overflow (1.3.2.8) and rounding (1.3.3.3).
- **The data bottleneck:** The CPU can sit idle while waiting for data in memory.
- **The parallelism limit:** Not every problem can be parallelized equally. If a step of a job depends on the result of the previous one, as in the ripple-carry adder, the steps cannot be done at the same time.
- **Security:** Because programs and data share the same system resources, programs must be isolated from one another. The strength of the stored-program idea, that data can be run like instructions, is also the door to malicious software.

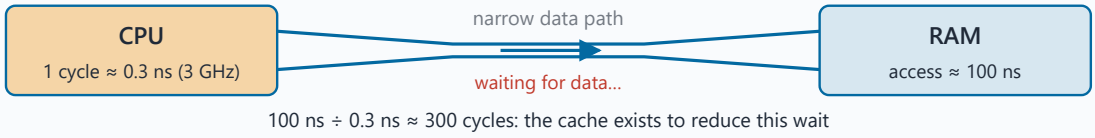


Figure 1.3.33: The data bottleneck. In a 3 GHz CPU one clock cycle takes about 0.3 ns; a RAM access takes about 100 ns. While waiting for a piece of data, the CPU can waste about 300 cycles.

The numbers in Figure 1.3.33 show the size of the bottleneck. A 3 GHz processor makes $3 \cdot 10^9$ cycles per second; one cycle takes $1 / (3 \cdot 10^9)$ s ≈ 0.33 ns. A 100 ns RAM access therefore corresponds to about 100 ÷ 0.33 ≈ 300 cycles. Without a cache the processor would spend most of its time waiting.

1.3.8.6 Performance: instruction count, CPI, and clock speed

The question of how fast a processor is cannot be answered with a single number. The time a program spends on the CPU is the product of three factors:

> CPU time = instruction count × CPI ÷ clock speed

- **Instruction count:** the number of machine instructions the program executes. It depends on the algorithm, the compiler, and the instruction set.

- **CPI** (cycles per instruction): how many clock cycles an instruction takes on average. It depends on the processor's internal design and on memory waits (cache misses).
- **Clock speed:** the number of cycles the processor makes per second; 3 GHz means $3 \cdot 10^9$ cycles per second. It depends on the manufacturing technology and on power and heat limits.

Case	Clock speed	CPI	Time
Processor A	3 GHz	1.5	$2 \cdot 10^9 \times 1.5 \div (3 \cdot 10^9) = 1.0$ s
Processor B	4 GHz	2.5	$2 \cdot 10^9 \times 2.5 \div (4 \cdot 10^9) = 1.25$ s
A, with a better compiler ($1.6 \cdot 10^9$ instructions)	3 GHz	1.5	$1.6 \cdot 10^9 \times 1.5 \div (3 \cdot 10^9) = 0.8$ s

Table 1.3.20: CPU time of a program of $2 \cdot 10^9$ instructions in three cases.

Example 1.3.13

Let us compute the times of processors A and B step by step.

1. **Total cycles (A):** $2 \cdot 10^9$ instructions × 1.5 cycles = $3 \cdot 10^9$ cycles.
2. **Time (A):** $3 \cdot 10^9$ cycles ÷ ($3 \cdot 10^9$ cycles/s) = 1.0 s.
3. **Total cycles (B):** $2 \cdot 10^9 \times 2.5 = 5 \cdot 10^9$ cycles.
4. **Time (B):** $5 \cdot 10^9 \div (4 \cdot 10^9) = 1.25$ s.
5. **Comparison:** 1.25 ÷ 1.0 = 1.25. Although B's clock speed is higher, A is 1.25 times faster.

The third row of the table shows that a program can speed up even with no change in hardware. If a better compiler brings the same program down to $1.6 \cdot 10^9$ instructions, A's time falls to 0.8 s. So a single number, for example clock speed alone, does not determine performance. The most reliable way to evaluate a processor is to run real programs on it and measure the time.

Common mistake

Thinking that the processor with the higher GHz value is always faster. Clock speed is only one factor of the equation. Even if a cycle takes less time, if more cycles are needed per instruction (a high CPI), the total time can get longer.

1.3.9 Levels of programming abstraction

1.3.9.1 From problem to circuit

When a programmer writes `total = a + b;` in Java, they do not think about transistors, registers, or opcodes. What makes this possible is that computer systems are built in layers. Each layer hides the detail of the one below and offers a more convenient view to the one above. Hiding the detail and

showing only what is needed is called **abstraction**.

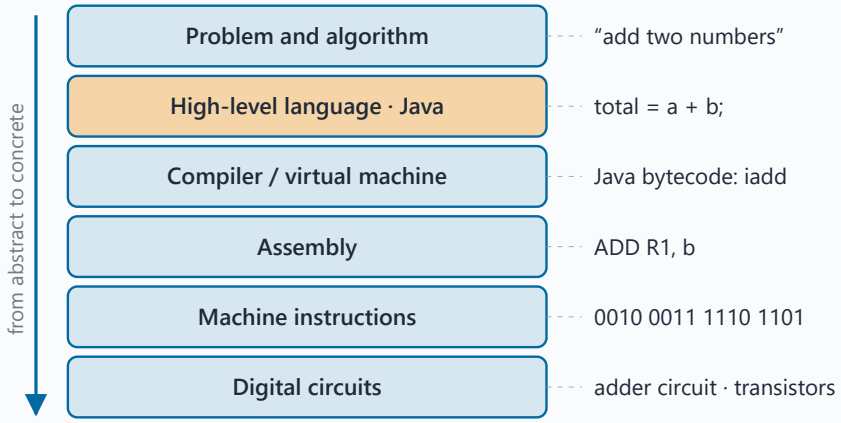


Figure 1.3.34: Levels of abstraction, from abstract to concrete, top to bottom: problem and algorithm ("add two numbers"), high-level language (`total = a + b;`), compiler or virtual machine (Java bytecode `iadd`), assembly (`ADD R1, b`), machine instructions (0010 0011 1110 1101), and digital circuits (adder, transistors). The example lines are for illustration.

Figure 1.3.34 shows the forms the same idea, "add two numbers," takes through the layers.

- **Problem and algorithm:** What is to be done and in which steps, in human language.
- **High-level language:** Languages such as Java, Python, and C. It is readable and **portable**: the same Java code runs on different processors.
- **Compiler or virtual machine:** Translates the high-level code to a lower level. The Java compiler first turns the code into bytecode (intermediate code); the Java virtual machine converts this code into the instructions of the processor it runs on.
- **Assembly:** Machine instructions written with names a human can read (LOAD, ADD). Each assembly instruction usually corresponds to a single machine instruction.
- **Machine instructions:** The bit sequences the processor executes directly. Assembly and machine code are specific to a particular processor architecture: machine code written for ARM does not run on x86-64.
- **Digital circuits:** The gates, adders, and transistors that actually execute the instruction.

The value of abstraction is that a person working at each layer needs to know only their own layer and the view offered by the layer just below. An application programmer can write correct programs without knowing the internal structure of the adder. Still, getting to know the lower layers helps one understand the causes of "unexpected" behavior such as overflow and rounding.

1.3.9.2 Tracing code: total = a + b

What a single line of Java turns into on the processor can be shown with a register-based assembly language. In memory there are `a = 12` and `b = 8` :

```
// Java: int total = a + b;
LOAD  R1, a      ; R1 ← a
ADD   R1, b      ; R1 ← R1 + b
STORE R1, total  ; total ← R1
```

Instruction	What happens?	R1	a	b	total
start	—	0	12	8	0
`LOAD R1, a`	the value of `a` is moved from memory to the register	12	12	8	0
`ADD R1, b`	`b` is read, the ALU computes 12 + 8 = 20, the result is written to R1	20	12	8	0
`STORE R1, total`	the 20 in R1 is written to the place of the `total` variable	20	12	8	20

Table 1.3.21: Assembly trace of the line `total = a + b`.

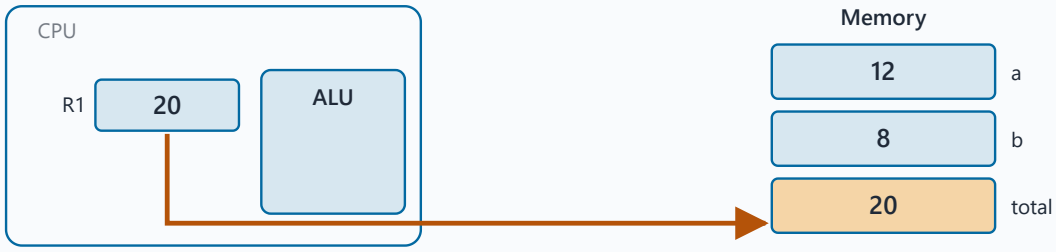


Figure 1.3.35: The STORE step. The 20 in R1 is written to memory, to the place of the `total` variable; the Java line is complete.

A single Java line has turned into three machine instructions. First the value is loaded from memory into a register, because the ALU works only with registers. Then the calculation is done, and last the result is written back to memory. This load–compute–store pattern appears in almost every assignment line on RISC-style processors. With the same template, the line `int diff = a - b;` turns into the instructions `LOAD R1, a`, `SUB R1, b`, `STORE R1, diff`. In subtraction the order of the operands matters: `a` is loaded first, then `b` is subtracted from it.

This small example brings together all the concepts at the beginning of the notes. `a` and `b` are two's complement bit sequences in memory; the ADD instruction sits in memory as a 16-bit number; the processor takes it with the fetch–decode–execute cycle; the chain of full adders does the addition; the result is written to memory over the data bus. The single line the programmer wrote runs by passing through all of these layers.

1.3.10 Summary

- A computer represents all data with **bit sequences**. n bits give 2^n different sequences. The **encoding** and the **data type** determine the meaning of a sequence; the same 01000001 is both 65 and "A."

- In a positional system, the position weights are powers of the base. Decimal is converted to binary by repeated division by 2. Since $8 = 2^3$ and $16 = 2^4$, binary is converted to octal and hexadecimal by grouping bits in threes and fours from the right.
- Negative integers are stored in **two's complement**: the bits are flipped and 1 is added. The range for n bits is $-2^{n-1} \ldots 2^{n-1} - 1$; in 8 bits, −128 … 127. Subtraction is done by the addition circuit as $a + (-b)$.
- If the result does not fit in the register, **overflow** occurs and the high bits are discarded: in 8 bits, 200 + 100 = 44. **Carry** is a problem of the unsigned interpretation, **signed overflow** of the signed one; if the sum of two numbers of the same sign comes out with the opposite sign, there is signed overflow.
- **Floating-point** numbers are stored according to IEEE 754 with a sign field, a (biased) exponent field, and a fraction field: `float` 1 + 8 + 23 bits, `double` 1 + 11 + 52 bits. Fractions whose denominator is not a power of 2 (such as 0.1) cannot be stored exactly; floating-point results are compared with a tolerance. Binary representation is exact for integers, while floating point is approximate for some decimals.
- **ASCII** is a 7-bit code (128 characters). **Unicode** gives each character a **code point**; **UTF-8** writes this number in 1–4 bytes: ASCII 1, Turkish letters 2, € 3, emoji 4 bytes. The number of characters and the number of bytes may differ.
- Bit operations (AND, OR, XOR, NOT, shifts) change individual bits with masks; a left shift multiplies by 2, and a right shift divides positive numbers by 2 (rounding down).
- **Logic gates** are the building blocks of circuits such as adders. The half adder produces SUM = A XOR B and CARRY = A AND B; the full adder adds the incoming carry as well. Multibit addition is done with a chain of full adders.
- In the **stored-program** approach, instructions and data are kept in the same memory as numbers. The CPU executes the instructions in the **fetch–decode–execute cycle** with the help of the PC and IR registers. The address, control, and data buses connect the CPU to memory. Branch instructions change the PC in the execute stage and so build loops and decisions.
- Memory is byte-addressed; multibyte values are written in **big-endian** or **little-endian** order.
- In the **von Neumann** architecture, instructions and data are in shared memory and on a shared path; this causes the von Neumann bottleneck. The **Harvard** architecture separates the memories; today's processors combine the two. RISC works with few, simple instructions and CISC with many, complex ones.
- CPU time = instruction count × CPI ÷ clock speed; no single number determines performance.
- High-level languages **abstract away** hardware details: a single Java line turns into a few machine instructions.

Calculation	Method	Example
Decimal → binary	divide by 2 repeatedly, read the remainders from last to first	$45 = 101101_2$

Calculation	Method	Example
Binary → octal / hexadecimal	group in 3s / 4s from the right	$101101 = 55_8 = 2D_{16}$
Negative (two's complement)	flip the bits, add 1	−5 = 11111011
Overflow (8-bit unsigned)	subtract 256 from the result	200 + 100 → 44
Fraction → binary	multiply by 2 repeatedly, take the integer parts	$0.375 = 0.011_2$
IEEE 754 `float`	sign, exponent + 127, fraction without the leading 1	$-6.25 = C0C80000_{16}$
UTF-8 (2 bytes)	11 bits → `110xxxxx 10xxxxxx`	"ğ" = C4 9F
CPU time	instruction count × CPI ÷ clock speed	$2 \cdot 10^9 \times 1.5 \div 3$ GHz = 1 s

Table 1.3.22: Kinds of calculations solved in these notes and their shortcuts.

1.3.11 Key terms

bit / byte encoding data type base (radix) binary / octal / decimal / hexadecimal signed / unsigned one's complement two's complement overflow signed overflow carry floating point IEEE 754 sign / exponent / fraction bias precision rounding ASCII control character bitwise operation bit mask bit shift Unicode code point UTF-8 string code unit field / record / file format logic gate half adder full adder ripple-carry adder stored program fetch–decode–execute cycle program counter (PC) instruction register (IR) accumulator (ACC) register control unit cache data / address / control bus memory address endianness branch von Neumann architecture von Neumann bottleneck Harvard architecture opcode instruction format RISC / CISC cycles per instruction (CPI) abstraction assembly language machine instruction

1.3.12 Test Yourself

1. How is 156 written in binary, octal, and hexadecimal?

Answer

Since 156 = 128 + 16 + 8 + 4, it is 10011100_2. Groups of three: 010 | 011 | 100 → 234_8. Groups of four: 1001 | 1100 → $9C_{16}$. Check: $9 \cdot 16 + 12 = 156$.

2. How is −20 written in 8-bit two's complement? Which number is the sequence 10110100?

Answer

+20 = 00010100; flip the bits: 11101011; add 1: 11101100 = −20. The sign bit of 10110100 is 1, so the number is negative. Flipping the bits and adding 1 gives 01001100 = 76; the sequence is −76. Check: −128 + 32 + 16 + 4 = −76.

3. In 8 bits, are there a carry and signed overflow in the operations 90 + (−30) and 70 + 80? Are the results correct?

Answer

90 + (−30): 01011010 + 11100010 = 1 00111100. The carry is 1 and is discarded; the remaining 00111100 = 60 is correct. The sum of two numbers with different signs does not overflow. 70 + 80: 01000110 + 01010000 = 10010110. The carry is 0, but the sum of two positives looks negative: there is signed overflow. The sequence reads as −106; the correct result, 150, does not fit in the signed 8-bit range.

4. What are the binary expansions of 0.625 and 0.2? Which is stored exactly as a `double`?

Answer

0.625 × 2 = 1.25 → 1; 0.25 × 2 = 0.5 → 0; 0.5 × 2 = 1.0 → 1 and the fraction is 0: 0.101_2, stored exactly (5/8, the denominator is a power of 2). 0.2 × 2 = 0.4 → 0; 0.8 → 0; 1.6 → 1; 1.2 → 1; then the fraction is 0.2 again: $0.0011\ 0011\ldots_2$, it repeats forever and cannot be stored exactly.

5. What is the 32-bit `float` form of 0.75 in hexadecimal?

Answer

$0.75 = 0.11_2 = 1.1 \times 2^{-1}$. The sign is 0; the exponent is −1 + 127 = 126 = 01111110; the fraction is 1 followed by twenty-two 0s. The bits: 0 01111110 1000…0 → 0011 1111 0100 0000 … → $3F400000_{16}$.

6. What are the UTF-8 bytes of the letter "ü" (U+00FC)? How many characters and how many bytes is the word "Gün" (Turkish for "day")?

Answer

$00FC_{16} = 252 = 11111100_2$; it does not fit in 7 bits, and in 11 bits it is written 00011111100. The 5 + 6 split: 00011 | 111100. The bytes are 110 00011 = C3 and 10 111100 = BC: "ü" = C3 BC. "Gün" is 3 characters; G and n are 1 byte each and ü is 2, so it is 4 bytes.

7. 0101 + 0011 are added in a chain of four full adders. What is the carry out at each position, and what are the result and the final carry?

   Answer

   i = 0: 1 + 1 + 0 = 10_2, digit 0, carry 1. *i = 1*: 0 + 1 + 1 = 10_2, digit 0, carry 1. *i = 2*: 1 + 0 + 1 = 10_2, digit 0, carry 1. *i = 3*: 0 + 0 + 1 = 1, digit 1, carry 0. The result is 1000_2 = 8 = 5 + 3; because the final carry is 0, it fits in 4 bits.

8. In the branching program, if the count at address 1010 were 3 instead of 2, what would `sum` be when the program ends? How many instructions would be fetched?

   Answer

   The loop turns 3 rounds and sum = 3 + 2 + 1 = 6. Each round is 8 instructions; at the exit LOAD, JUMPZ, and HALT are fetched: 8 · 3 + 3 = 27 instructions.

9. How many seconds does a program of $3 \cdot 10^9$ instructions take on a 2 GHz processor with CPI 2 and on a 3 GHz processor with CPI 1.2? How many times faster is the second?

   Answer

   First: $3 \cdot 10^9 \times 2 \div (2 \cdot 10^9) = 3$ s. Second: $3 \cdot 10^9 \times 1.2 \div (3 \cdot 10^9) = 1.2$ s. 3 ÷ 1.2 = 2.5; the second processor is 2.5 times faster.

10. How is the 32-bit value 0x0A0B0C0D written to addresses 200–203 on a little-endian machine? If a big-endian machine reads these four bytes in one go, what value does it see?

    Answer

    In the little-endian arrangement the least significant byte goes to the lowest address: 200: 0D, 201: 0C, 202: 0B, 203: 0A. A big-endian machine takes the first byte as the most significant and sees 0x0D0C0B0A.

1.3.13 Further reading

- C. Petzold, *Code: The Hidden Language of Computer Hardware and Software* (2nd ed., Microsoft Press, 2022): starting from Morse code and circuits with light bulbs, it builds gates, adders, memory, and a simple processor step by step without requiring mathematical background.
- N. Nisan and S. Schocken, *The Elements of Computing Systems* (2nd ed., MIT Press, 2021): starting from the NAND gate and having the reader build all the layers up to the ALU, the CPU, the assembler, and the compiler, it works through the levels of abstraction in these notes in a hands-on way.

- R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective* (3rd ed., 2016), Chapter 2: explains the two's complement representation of integers, overflow, and IEEE 754 from the programmer's viewpoint and with many examples.
- D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys* 23(1), 1991: the classic article that treats in detail the causes and consequences of rounding errors in floating-point computation.

1.3.14 References

1. D. A. Patterson, J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann.
2. R. E. Bryant, D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3rd ed., Pearson, 2016.
3. The Unicode Consortium, *The Unicode Standard*, https://www.unicode.org/standard/standard.html.
4. IEEE, *IEEE Standard for Floating-Point Arithmetic* (IEEE Std 754-2019), 2019.
5. C. Petzold, *Code: The Hidden Language of Computer Hardware and Software*, 2nd ed., Microsoft Press, 2022.
6. N. Nisan, S. Schocken, *The Elements of Computing Systems: Building a Modern Computer from First Principles*, 2nd ed., MIT Press, 2021.
7. D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, 23(1), 5–48, 1991.

PART I · CHAPTER 1.4

The Internet

Opening a web page, sending an email, or joining a video call now takes a few seconds. Behind each of these actions are millions of interconnected networks, devices that speak common rules, and servers on the other side of the world. These notes follow that arrangement from end to end. They begin with what a computer network is, how you connect to the internet, the difference between bandwidth and latency, and the devices that make up a network. They then work step by step through how data is packaged in layers, how packets pass through routers, and how IP addresses, NAT, DNS, URLs, ports, and the TCP connection work. After encryption and HTTPS, they turn to how the Web, search engines, cloud services, and email work. The last part is about security: the three goals of information security, malware, phishing, password strength, multi-factor authentication, VPNs, firewalls, and backups.

LEARNING OUTCOMES

Network concepts

Explain the relationship among the internet, clients, servers, and protocols; distinguish the roles of network devices; do simple timing calculations with bandwidth and latency.

Addressing

Tell IP addresses, domain names, DNS, URLs, and ports apart; convert an IPv4 address to binary; trace a packet through the layers, the routers, and NAT.

Using the Web

Describe the journey of a web request step by step; explain how search engines work with an inverted index; use search operators and criteria for evaluating sources.

Security

Recognize threats; spot the signs of a phishing message; compare password strength by calculation; apply measures such as multi-factor authentication, firewalls, and backups where they fit.

1.4.1 Network fundamentals

The internet is not a single machine or a single cable. It is a large number of independently built networks that connect to one another by following common rules. To understand how the internet works, therefore, start by understanding how a small network works. This section sets out the basic concepts: network, internet, access line, bandwidth, latency, client, server, protocol, and network devices. It ends with wireless networks and their security.

1.4.1.1 Computer networks and the internet

Picture a home with two laptops, a phone, and a printer. One laptop wants to send a document to the printer, and the other wants to pull photos off the phone. For these devices to exchange data, they must be connected somehow. The connection may be a cable or a radio wave. Whatever form the connection takes, the resulting structure is called a network.

Definition 1.4.1

A **computer network** is a structure formed by connecting devices so that they can share data and resources. The shared resource may be a file, a printer, an internet connection, or a service.

A home network is small; a university network contains hundreds of buildings and thousands of devices; a company network links branches in different cities. Each of these networks is built and managed by its own owner. The internet makes these separate networks work as one giant network.

Definition 1.4.2

The **Internet** is the global **network of networks** that joins networks of different sizes under common rules. These common rules are called **TCP/IP**: any network that follows them can become part of the internet.

Figure 1.4.1 shows four separate networks: a home network, a university network, a company network, and a data center. Each network has its own devices, and each connects to the outside through a **router**. The routers in turn attach to the **internet backbone**, the large interconnected lines of the internet service providers. To reach the video server in the data center, the phone on the home network passes through at least two routers and the backbone.

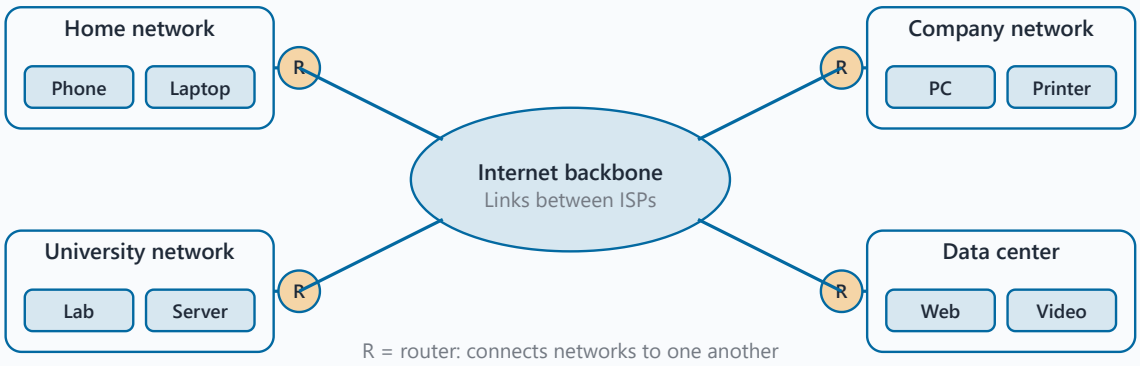


Figure 1.4.1: A network of networks. Home, university, and company networks and a data center connect to the internet backbone through their own routers (R). The backbone consists of the high-capacity links between internet service providers.

An important consequence follows: the internet has no center. When one network goes down, the others keep working; when a line is cut, traffic can often take another route. The internet was designed with exactly this resilience in mind.

Common mistake

Thinking the internet is one network run by one company. The internet consists of networks run by thousands of independent organizations. What holds them together is not a common owner but the common protocols that everyone follows.

1.4.1.2 The development of the internet

The internet's roots go back to the late 1960s. **ARPANET**, supported by a research agency of the US Department of Defense, made its first connections in 1969, between computers at the University of California, Los Angeles (UCLA) and the Stanford Research Institute (SRI). In the following years new universities and research centers joined the network. Because each new network worked by its own rules, communication between networks was difficult. The problem was solved on 1 January 1983, when ARPANET switched to **TCP/IP** as its common protocol. Today's internet grew on top of that common language.

In 1989 Tim Berners-Lee, working at CERN, proposed an information system that linked documents to one another with links. That proposal was the beginning of the **Web** (the World Wide Web). The first website went live in 1991 and the Web was opened to everyone. In 2007 touchscreen smartphones began to spread, and the internet moved into the pocket. Today mobile access has overtaken desktop access.

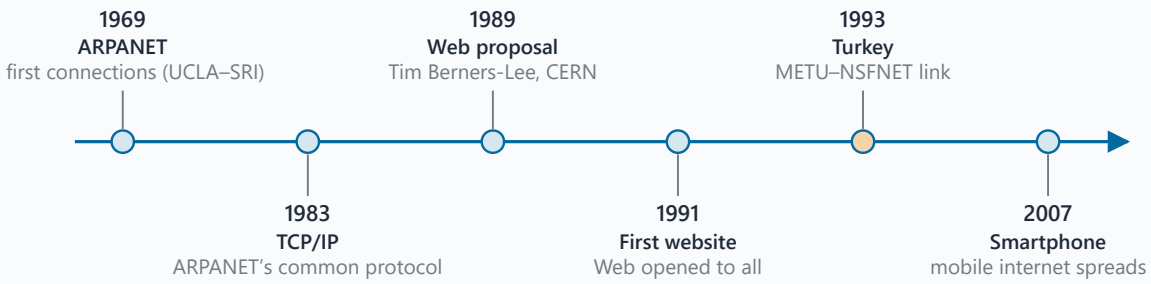


Figure 1.4.2: Six milestones in the development of the internet: ARPANET's first connections (1969), the switch to TCP/IP (1983), the Web proposal (1989), the first website (1991), Turkey's connection to the internet (1993), and the smartphone (2007).

Note that the two concepts have separate histories. The internet is an **infrastructure** that has been developing since 1969; the Web is a **service** that began in 1991 on top of that infrastructure. The difference between the two is covered in detail in 1.4.5.1.

The internet in Turkey. Turkey's first permanent internet connection was established on 12 April 1993 between Middle East Technical University (METU) and the US academic backbone NSFNET. The line ran at 64 Kbps, that is, 64 thousand bits per second. In 1996 the National Academic Network and Information Center (ULAKBİM) was founded within TÜBİTAK. ULAKNET, the national academic network linking universities to one another and to the internet, went into operation in 1997. Today fiber, mobile, and satellite connections offer different access options, and fiber lines that run all the way to homes reach speeds of 1 Gbps.

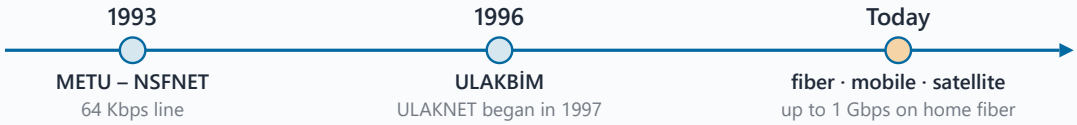


Figure 1.4.3: Timeline of the internet in Turkey: the 64 Kbps METU–NSFNET line in 1993, the founding of ULAKBİM in 1996, ULAKNET in 1997, and fiber, mobile, and satellite access today.

1.4.1.3 Connecting to the internet

The network of a home or an organization cannot connect to the internet on its own; a service provider steps in between. The provider connects the user's network to its own network, and its own network to the internet backbone.

Definition 1.4.3

An **Internet service provider** (ISP) is the organization that connects a user to the internet backbone. The connection between the user's network and the ISP is called the **access line**.

Figure 1.4.4 shows the path from a home to the internet. Devices at home connect to the modem/router by Wi-Fi or cable. That box connects to the ISP over the access line, and the ISP connects to the backbone. The caption under the figure names both kinds of network: the inside of the home is a local area network (LAN), and the outside is a wide area network (WAN). These two terms are defined in 1.4.1.7.

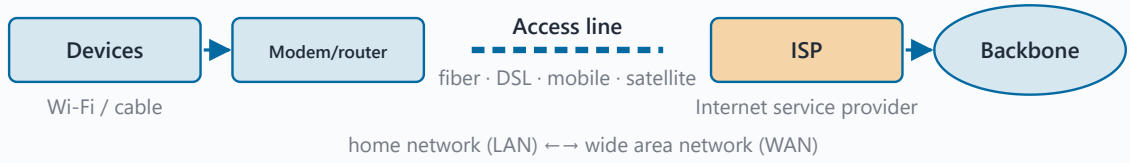


Figure 1.4.4: From home to the internet: devices, modem/router, access line (fiber, DSL, mobile, or satellite), internet service provider, and backbone.

The access line can be built with different technologies. Table 1.4.1 compares four common options.

Type	How it works	Strength	Weakness
Fiber	Data travels as pulses of light inside glass fibers.	High capacity and low latency.	The cable has to be laid all the way to the building.
DSL	Uses the existing copper telephone line.	Available almost anywhere there is a telephone line.	Speed drops as the distance to the exchange grows.
Mobile data	A phone or modem talks to base stations over radio waves.	Portable; needs no cable.	Speed depends on coverage and crowding; often capped by a data quota.
Wireless and satellite	Uses a fixed wireless transmitter or an orbiting satellite.	An option in rural and mountainous areas where laying cable is hard.	Latency is very high with a high-orbit satellite.

Table 1.4.1: Types of internet access lines.

In homes, the modem, router, wireless access point, and a small switch are often combined in a single box. In everyday speech this box is called “the modem,” but several separate devices actually run inside it. The roles of these devices are described one by one in 1.4.1.6.

1.4.1.4 Bandwidth and latency

When we say a connection is “fast,” we may actually mean either of two separate properties. The first is how much data the connection can carry per second. The second is how long the first bit sent takes to arrive at the other end. The two properties are independent, and each is decisive for different jobs.

Definition 1.4.4

Bandwidth is the maximum amount of data a connection can carry per unit of time, measured in bits per second (bit/s, Mbit/s, Gbit/s). **Latency** is the time a bit takes to get from source to destination, measured in milliseconds (ms). Round-trip latency is what people commonly call the **ping** value.

It helps to think of the two concepts as a water pipe (Figure 1.4.5). Bandwidth is the pipe's *width*: a wide pipe delivers more water per second. Latency is the pipe's *length*: in a long pipe, water takes longer to come out of the far end after you open the tap. A pipe can be very wide and very long, or narrow and short.

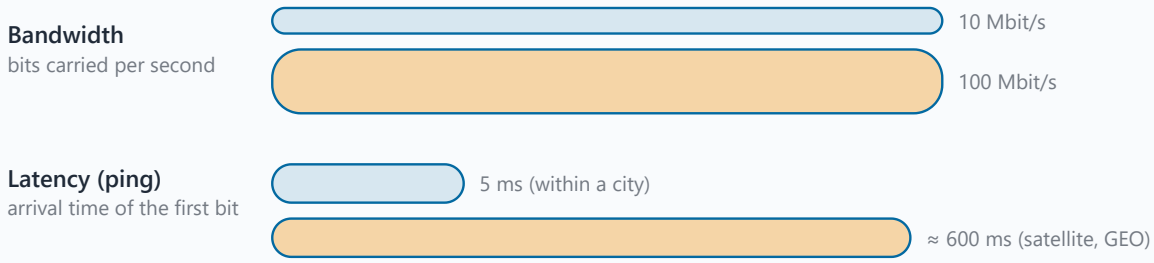


Figure 1.4.5: Bandwidth is like the width of a pipe and latency like its length. The difference between 10 Mbit/s and 100 Mbit/s shows up in the width; the difference between 5 ms within a city and about 600 ms on a geostationary (GEO) satellite shows up in the length.

Bits and bytes. Units are the most common source of confusion in these calculations. Line speeds are given in *bits* (Mbit/s or Mbps). File sizes are given in *bytes* (MB, GB). A byte is 8 bits. So when computing a download time, first convert both quantities to the same unit. In these notes 1 GB = 1000 MB and 1 MB = 1000 KB.

> download time (s) = file size (MB) × 8 ÷ line speed (Mbit/s)

This formula gives the "best case": it assumes the line runs at full speed, nobody else is using it, and protocol headers add no overhead. The real time is somewhat longer.

Example 1.4.1

On a 100 Mbit/s line, how many seconds does a 500 MB file take to download in the best case?

1. Convert the line speed to bytes: 100 Mbit/s ÷ 8 = 12.5 MB/s. The line carries 12.5 megabytes per second.
2. Divide the file size by this rate: 500 MB ÷ 12.5 MB/s = 40 s.

3. The other route gives the same result: 500 MB = 500 × 8 = 4000 Mbit; 4000 ÷ 100 = 40 s.

In the best case the file downloads in **40 seconds**. By the same method, a 1.5 GB file on a 200 Mbit/s line gives 1.5 GB = 1500 MB = 12,000 Mbit, and 12,000 ÷ 200 = 60 s.

Common mistake

People take a "100-megabit internet connection" to mean "100 megabytes per second." A 100 Mbit/s line carries at most 12.5 MB per second. Mixing up bits and bytes makes the result off by exactly a factor of 8.

Where does latency come from? A signal cannot travel faster than light. In fiber, light moves through glass more slowly than through a vacuum, at about 200,000 km per second. So a round trip by fiber to a server 800 km away takes at least 2 × 800 ÷ 200,000 s = 8 ms. Even with no other waiting at all, this time cannot get shorter. Queuing and processing times in routers add to it.

Example 1.4.2

Geostationary (GEO) satellites orbit about 35,786 km above the equator. What is the lower bound on round-trip latency for a connection through a satellite? (Speed of light ≈ 300,000 km/s.)

1. In one direction, the signal goes up from the ground to the satellite and down from the satellite to the ground: 2 × 35,786 = 71,572 km.
2. The time for this path: 71,572 ÷ 300,000 ≈ 0.239 s, or about 239 ms.
3. The request goes out and the reply comes back: a round trip covers the path twice, so the time is about 477 ms.

The path alone takes about half a second. Add the processing at the ground stations and the terrestrial links, and you reach the measured value of **≈ 600 ms**. This latency does not go away, however high the line's bandwidth.

Which one matters? The answer depends on the kind of job. When downloading a large file, bandwidth is what counts: the first bit arriving half a second late goes unnoticed in a download that lasts a few minutes. In gaming, video calls, and remote desktop connections, low latency is what counts. A video call is content with a few Mbit/s, but at 600 ms latency the speakers talk over each other. Compare, for example, a 100 Mbit/s satellite line with a 30 Mbit/s fiber line. A 1 GB file downloads in about 80 s on the satellite line and about 267 s on the fiber line. For a video call, though, the fiber line with about 10 ms latency is clearly better.

1.4.1.5 Clients, servers, and protocols

Most communication on the internet follows the same pattern. One side asks for something, and the other side fulfills the request. For example, a student's browser asks for a course page; the university's web server sends the page. The weather app on a phone asks for a forecast; the weather service's server sends it.

Definition 1.4.5

A **client** is a device or program that requests a service. A **server** is a system that fulfills requests and provides resources. A **protocol** is the set of rules that fix the form and order of communication: who speaks first, in what format a message is written, and what to do if an error occurs.

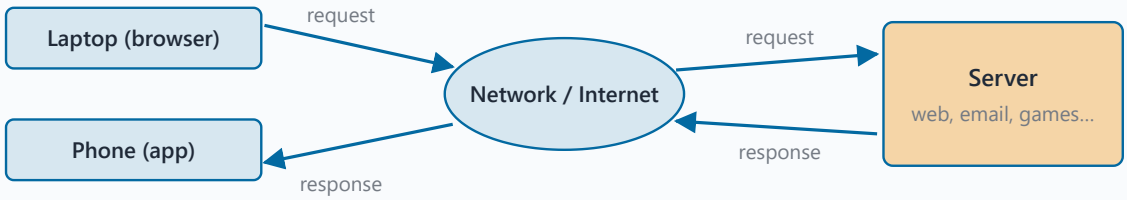


Figure 1.4.6: Clients (a laptop running a browser, a phone running an app) send requests to the server over the network; the server returns a response to each request.

The word "server" is used in two senses. Sometimes it means hardware, a powerful computer in a data center. Sometimes it means a program, the software that listens for requests and answers them. A web server, an email server, and a database server can all run on the same machine. Likewise, a computer can be a client for one job and a server for another.

1.4.1.6 Network devices

A network consists of devices plus the special hardware that connects them to one another and to the outside world. Figure 1.4.7 shows the order of equipment in a typical home network: the line from the internet reaches the modem first, and from there the firewall and the router. Behind the router are a switch for wired devices and an access point for wireless devices.

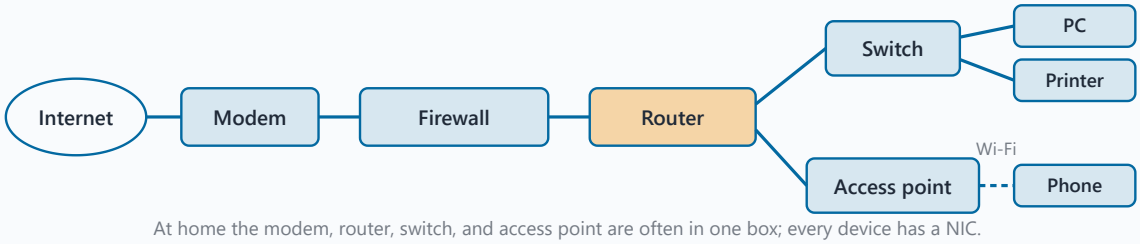


Figure 1.4.7: Home network diagram: internet, modem, firewall, router; from the router through a switch to a PC and a printer, and from the access point over Wi-Fi to a phone. Every device has a network interface card (NIC).

Device	Role	What does it go by?
Network interface card (NIC)	Connects the device to the network; puts bits onto the cable or radio wave.	Every card has a MAC address assigned at manufacture.
Switch	Forwards frames within the local network only to the destination device.	The destination MAC address in the frame.
Router	Routes packets between different networks.	The destination IP address in the packet and the routing table.
Modem	Converts the signal on the access line (light, electricity, radio) to network data, and back.	The physical signal of the line.
Access point	Connects wireless devices to the local network.	The Wi-Fi standard and channel settings.
Firewall	Inspects network traffic against rules; allows or blocks it.	Mostly the IP address and port number.

Table 1.4.2: Network devices and their roles.

Two devices in the table are often confused: the switch and the router. The difference is which boundary they work within. A **switch** works inside a single local network. When the office PC sends data to the printer, the data reaches the switch; the switch knows which of its ports the printer is plugged into and delivers the data only to that port. A **router** sits at the border between networks. When the same PC wants to connect to a website, the data must leave the local network; the router does this and sends the data on toward the next network.

Common mistake

Thinking the modem does all the work because the box at home is called a “modem.” The modem only translates the signal of the access line. Assigning addresses, routing between networks, and NAT are the router’s job; the wireless connection is the access point’s; filtering traffic is the firewall’s. All of them may be in the same box, but their roles are separate.

1.4.1.7 A typical network and geographic scope

Put the devices together and a typical organizational network appears (Figure 1.4.8). Two PCs and a printer in the office connect to the switch by cable. The phone and tablet connect wirelessly through the access point, which is itself connected to the switch. The switch connects to the router, the router to the firewall, and the firewall to the ISP line through the modem. A branch of the same organization in another city is also connected to the internet with its own router. The two offices reach each other over the internet.

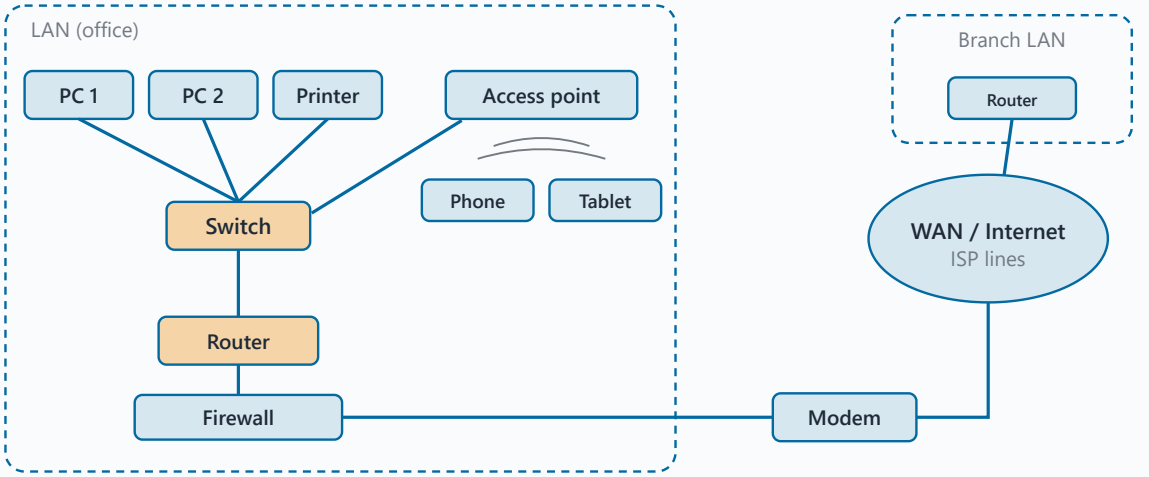


Figure 1.4.8: A typical network from LAN to WAN. The inside of the dashed frame is the office's local area network (LAN). The switch connects devices on the same local network; the router connects different networks. The branch network and the office network communicate over a wide area network (WAN) made up of ISP lines.

Networks are also named by their geographic scope. As the scope grows, building and running the network passes to other organizations (ISPs, operators).

Type	Stands for	Scope	Example
PAN	personal area network	A few meters	A phone and wireless earbuds
LAN	local area network	A room, building, or campus	An office network, a school lab
MAN	metropolitan area network	City scale	A municipal network
WAN	wide area network	Country or continent	An organization's branches; the internet is the largest WAN

Table 1.4.3: Geographic scope of networks.

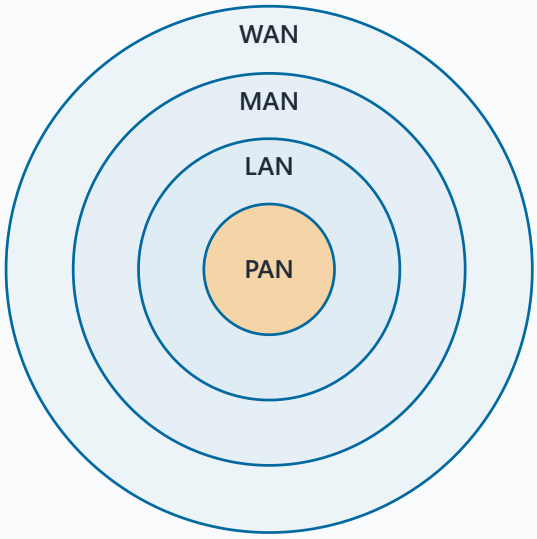


Figure 1.4.9: The scopes of networks are like nested rings: PAN innermost, followed by LAN, MAN, and WAN.

1.4.1.8 Wi-Fi standards and bands

Wireless local area networks work according to the IEEE (Institute of Electrical and Electronics Engineers) 802.11 family of standards. Because the technical names of the standards are hard to remember, manufacturers have used generation numbers since 2018: Wi-Fi 5, Wi-Fi 6, and so on. Cellular phone networks, by contrast, rest on the standards of a different body (3GPP).

Name	Standard	Bands	Main difference
Wi-Fi 5	802.11 ac	5 GHz	Wide channels; a clear speed increase over 802.11n.
Wi-Fi 6	802.11 ax	2.4 and 5 GHz	More efficient in crowded environments (OFDMA); better for networks with many devices.
Wi-Fi 6E	802.11 ax	2.4, 5, and 6 GHz	Adds the 6 GHz band to Wi-Fi 6; less interference.
Wi-Fi 7	802.11 be	2.4, 5, and 6 GHz	Even wider channels (320 MHz); simultaneous connections on several bands (MLO).
4G (LTE) → 5G	3GPP	Operator bands	5G typically offers higher speed and lower latency; actual values depend on the operator and coverage.

Table 1.4.4: Wi-Fi generations and mobile networks.

The two abbreviations in the table need a short explanation. **OFDMA** (orthogonal frequency-division multiple access) splits a channel into small sub-parts and shares them among several devices at once. Dozens of phones in a crowded classroom therefore do not wait in turn; each gets a small share at the same time. **MLO** (multi-link operation) lets a device connect on several bands at once; when one band is congested, data flows over another.

The speeds given for Wi-Fi standards are theoretical maximums. The actual speed depends on distance, obstacles in between, channel width, and which standard both the router and the device support. A router that supports Wi-Fi 7 talks to a laptop that supports only Wi-Fi 5 at Wi-Fi 5 speed.

Bands. Wi-Fi can work in three frequency bands: 2.4 GHz, 5 GHz, and 6 GHz. Each band has strengths and weaknesses (Figure 1.4.10).

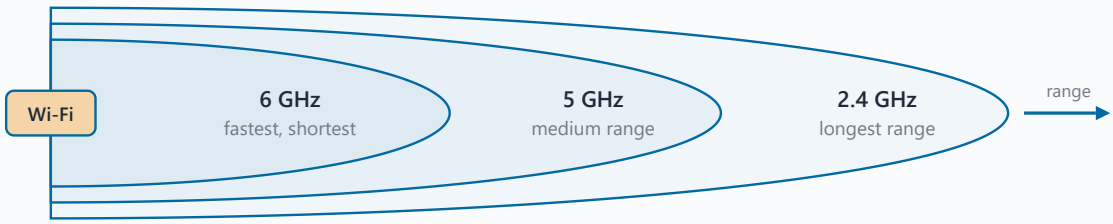


Figure 1.4.10: Range of the Wi-Fi bands. 6 GHz is the fastest but shortest-range band; 5 GHz has medium range; 2.4 GHz reaches farthest.

- **2.4 GHz:** Long range, and it passes through walls better; speed is typically low. It is crowded because microwave ovens, Bluetooth, and neighboring networks use the same band.
- **5 GHz:** Medium range, higher speed. It is attenuated more by walls than 2.4 GHz.
- **6 GHz:** Short range, the highest speed potential, and little crowding (Wi-Fi 6E and 7). It is the band most affected by obstacles.

The general rule is this: as frequency rises, speed potential rises while range and wall penetration fall. This trade-off has a physical cause. The wavelength of an electromagnetic wave is the speed of light divided by the frequency: $\lambda = c / f$. As frequency rises the wavelength shortens, and short waves lose more energy in obstacles such as walls, concrete, and water.

Example 1.4.3

What are the wavelengths of the three Wi-Fi bands? ($c \approx 3 \times 10^8$ m/s.)

1. 2.4 GHz: $\lambda = 3 \times 10^8 \div 2.4 \times 10^9 = 0.125$ m, or about 12.5 cm.
2. 5 GHz: $3 \times 10^8 \div 5 \times 10^9 = 0.06$ m, or 6 cm.
3. 6 GHz: $3 \times 10^8 \div 6 \times 10^9 = 0.05$ m, or 5 cm.

The 2.4 GHz wave is two and a half times as long as the 6 GHz wave. This difference explains why 2.4 GHz gives a steadier connection in a room two walls away.

The practical conclusion is simple: in a room far from the modem, 2.4 GHz is often the better choice; at close range, 5 or 6 GHz is. Many routers steer the device to the best band automatically.

1.4.1.9 Wi-Fi security

To join a wired network, you need physical access to the cable. On a wireless network, the signal spreads beyond the walls, into the street and neighboring apartments. So the traffic of a wireless network must be encrypted, and only those who know the password should be able to join. Wi-Fi security protocols do this job.

Protocol	Status	Description
WEP	Insecure	Very old; can be broken within minutes. It should no longer be used.
WPA / TKIP	Legacy	Not recommended; exists only for compatibility with old devices.
WPA2 (AES)	Common	A strong, long password is essential; a weak password can be guessed.
WPA3	Current	More resistant to weak passwords; on the 6 GHz band WPA3 (or Enhanced Open for password-free networks) is mandatory. Requires device support.

Table 1.4.5: Wi-Fi security protocols.



Figure 1.4.11: The order of the Wi-Fi security protocols, from left to right from old to new and from weak to strong: WEP, WPA, WPA2, WPA3.

An important difference between WPA2 and WPA3 is how they behave against password guessing. With WPA2, an attacker can record the short handshake traffic of a device joining the network and try passwords on their own computer without ever touching the network. If the password is short or a dictionary word, it is found quickly. With WPA3, every guess requires talking to the network again, so offline guessing is not possible.

Open (password-free) networks. On cafe, airport, or hotel networks, traffic may be unencrypted. Someone on the same network can eavesdrop on the traffic, or set up a fake access point with the same name and lure users to their own device. So on such networks, sensitive actions such as banking or e-Devlet should be done only over HTTPS, and over a VPN if necessary. HTTPS is covered in 1.4.4.2 and VPNs in 1.4.7.8.

Common mistake

Thinking that hiding the network name, or allowing only the MAC addresses of familiar devices, makes a network secure. A hidden network name and MAC addresses are both openly visible over the air and are easily spoofed. What protects a wireless network is a current protocol (WPA2 or WPA3) and a long, unguessable password.

1.4.2 Protocols, layers, and packets

For two computers to understand each other, they must follow common rules. But even a single job such as “fetch a web page” consists of many sub-tasks: the page is requested, the request is split into pieces, the pieces are numbered, each piece is addressed, a route is found, the bits are put on the wire, and lost pieces are resent. Rather than loading all of this onto one protocol, each sub-task is given to a separate protocol, and the protocols are stacked in layers. This section first introduces the core protocols, then follows the layer model, splitting into packets, and the passage of packets through routers step by step.

1.4.2.1 Core communication protocols

The six protocol families that will come up often are summarized in Table 1.4.6. The details of each come later in the notes; the aim here is to see up front which job falls to which protocol.

Protocol	Stands for	Role
IP	Internet Protocol	Addressing and routing packets between networks.
TCP	Transmission Control Protocol	Ordered, reliable delivery of data.
UDP	User Datagram Protocol	Low-latency, connectionless delivery.
DNS	Domain Name System	Converting a domain name to an IP address.
HTTP / HTTPS	Hypertext Transfer Protocol, and its secure version	Transfer of web resources; HTTPS encrypts the communication.
SMTP / IMAP	Simple Mail Transfer Protocol / Internet Message Access Protocol	Sending (SMTP) and reading (IMAP) email.

Table 1.4.6: Core communication protocols.

The choice between TCP and UDP is a trade-off. TCP checks that every packet arrived, resends any that were lost, and delivers the data in the right order. The price is extra messages and waiting. UDP sends the packet and does not look back: if there is loss, it does not make up for it. When downloading a file or a web page, even a single missing byte is unacceptable, so TCP is used. In a live video call, there is no point in resending a video frame lost half a second ago; a new frame has arrived by then. UDP is preferred for jobs like this. DNS queries are also mostly carried over UDP because they are short.

1.4.2.2 Protocols work layer by layer

Think of sending a letter by post. The person who writes the letter cares only about the content. The person who seals the envelope and writes the address does not know what the letter says. The post office puts the envelope in a bag and writes on the bag which city it is going to. The truck driver only

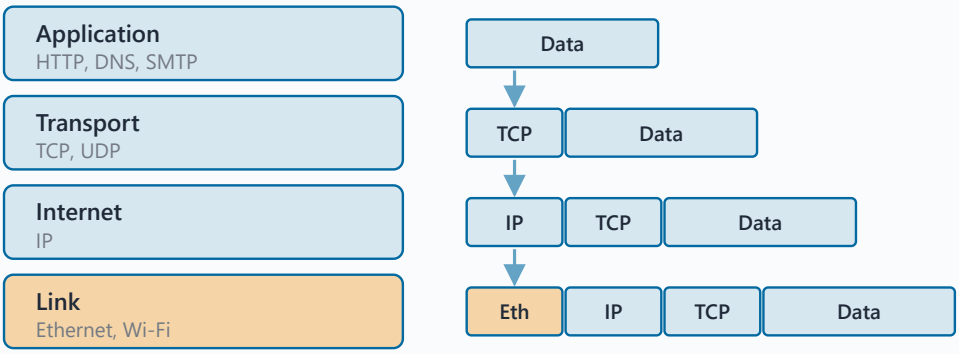
carries bags. Each stage does its own job, does not look inside the stage above it, and does not concern itself with how the stage below works. Network protocols are organized on the same principle.

Definition 1.4.6

Internet protocols are grouped into four **layers**. The **application layer** decides what is said (HTTP, DNS, SMTP). The **transport layer** splits data into pieces and manages end-to-end delivery (TCP, UDP). The **internet layer** addresses packets and routes them between networks (IP). The **link layer** physically delivers bits to the next device on the same network (Ethernet, Wi-Fi). Each layer adds its own **header** to the data it passes down; this process is called **encapsulation**.

On the sending side, data moves from top to bottom and gains a header at each layer. The upper part of Figure 1.4.12 shows this build-up:

1. **Application layer.** The browser produces an HTTP request, for example "`GET /duyurular`". As far as the lower layers are concerned, this request is just "data."
2. **Transport layer.** If necessary, the data is split into pieces, and a **TCP header** is added in front of each piece. The header contains the port number (which application the data is going to) and the sequence number (the piece's place in the whole). Thanks to this number, a lost piece can be requested again.
3. **Internet layer.** An **IP header** is added in front of the TCP segment: the source and destination IP addresses. Routers choose a route by looking only at this header; they do not need to read the TCP header or the HTTP request inside.
4. **Link layer.** A frame header is added on the outside. This header contains the **MAC address** of the next device on the local network. The bits are then put on the cable or radio wave.



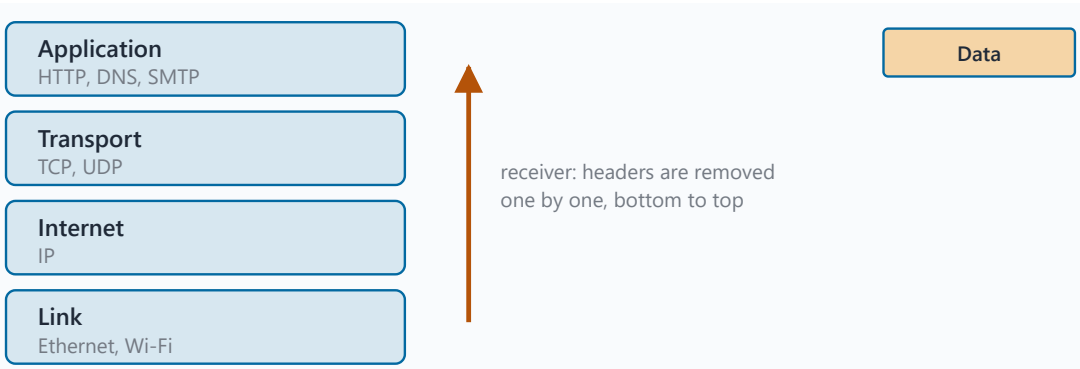


Figure 1.4.12: Encapsulation. At the top, the sender: the data gains a header at each layer and is finally sent as Eth | IP | TCP | Data. At the bottom, the receiver: the headers are removed one by one from bottom to top, and the data reaches the application.

On the receiving side the process runs in reverse. The link layer reads and removes the frame header. The internet layer looks at the IP header; if the destination address is this machine, it removes the header and passes the content up. The transport layer arranges the pieces by sequence number and, from the port number, finds which application the data belongs to. In the end, the web server receives exactly the request the browser produced.

Layer	Example protocols	What is in the header?	Who reads this header?
Application	HTTP, DNS, SMTP	The request itself (method, resource path)	The application on the other side
Transport	TCP, UDP	Source and destination port, sequence number	The operating systems at the ends
Internet	IP	Source and destination IP address, TTL	Every router along the way
Link	Ethernet, Wi-Fi	Source and destination MAC address	The next device on the same local network

Table 1.4.7: Headers of the layers and who uses them.

The biggest gain of the layered structure is independence. The browser does not need to know whether the data goes by fiber or by Wi-Fi. A router does not need to know whether a packet contains a web request or an email. When a new technology appears in one layer (Wi-Fi 7, for example), nothing changes in the other layers.

Encapsulation also has a cost: each header takes up a few dozen bytes. When the actual data is small, the share taken by headers grows.

Example 1.4.4

Typical header sizes are as follows: TCP 20 bytes, IPv4 20 bytes, Ethernet 18 bytes (a 14-byte header and 4 bytes of error checking). For a 100-byte HTTP request and for a 1460-byte piece of data, what are the total length on the wire and the efficiency?

1. Total of the headers: 20 + 20 + 18 = 58 bytes.
2. 100-byte request: length on the wire 100 + 58 = 158 bytes. Efficiency 100 ÷ 158 ≈ 63.3%.
3. 1460-byte piece: 1460 + 58 = 1518 bytes. Efficiency 1460 ÷ 1518 ≈ 96.2%.

In small messages, more than a third of the bytes on the wire are header. In a large file transfer, the headers' share falls below 4%. 1518 bytes is the usual maximum size of an Ethernet frame; that is why TCP data is mostly carried in 1460-byte pieces.

Common mistake

Thinking that a router reads the web page inside a packet and chooses a route accordingly. A router looks only at the IP header. What is inside the packet does not affect route selection; with HTTPS the content is encrypted anyway and the router cannot read it.

1.4.2.3 Data is split into packets

Large data is not sent in one piece. There are two reasons. First, a long transfer would occupy the line by itself for a while, and nobody else could get in. Second, if a single bit were corrupted along the way, all of the data would have to be resent. When data is split into small pieces, the line is shared among many users and only the single corrupted piece is resent.

Definition 1.4.7

A **packet** is a piece of data carried independently across the network. Each packet carries its own header: the source address, the destination address, and a sequence number. The piece that TCP carries is also called a **segment**. This arrangement, in which data is split into packets and sent, is called **packet switching**.

In the example, the message "HELLO, WORLD!" is sent. Counting the comma, the space, and the exclamation mark, the message is 13 characters, and it is split into four packets: "HELL", "O, W", "ORLD", and "!". Between the sender and the receiver there are four routers: R1, R2, R3, and R4. Two paths lead from R1 to R4: one through R2, the other through R3 (Figure 1.4.13).

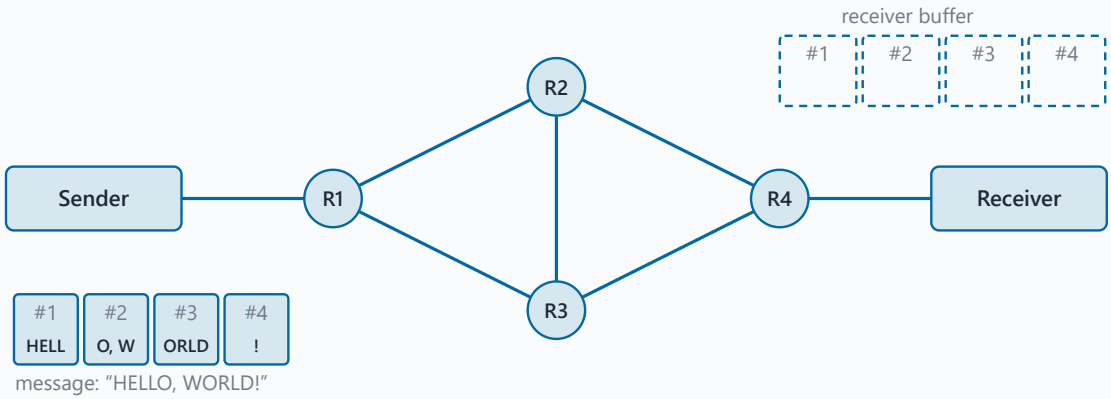
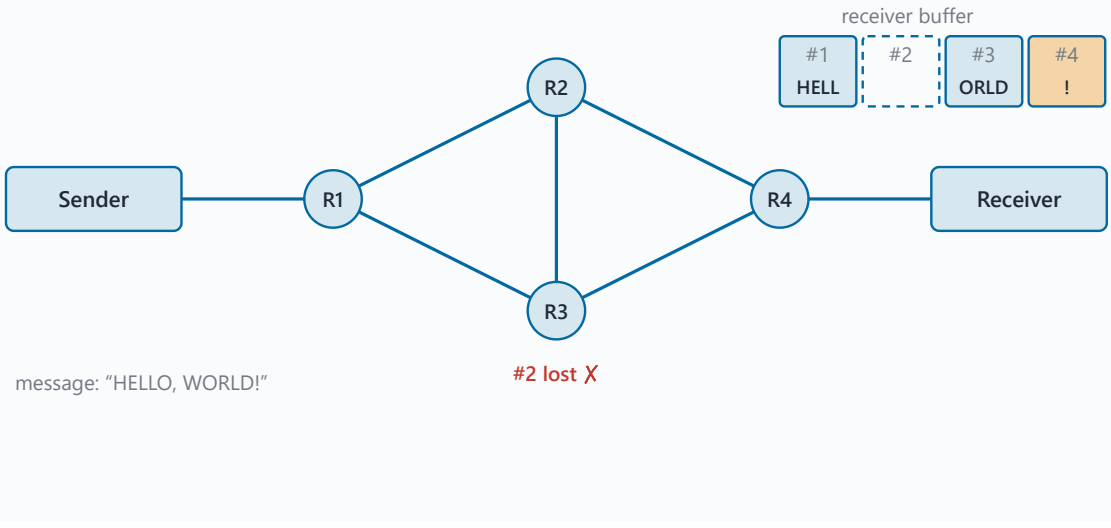


Figure 1.4.13: The message "HELLO, WORLD!" is split into four packets. At the top right the receiver's buffer is empty; as packets arrive, they take their places in this buffer according to their sequence numbers.

The journey of the packets is traced step by step in Table 1.4.8.

Step	Event	Receiver's buffer (#1 #2 #3 #4)
1	The message is split into four packets; each header has a sequence number, source address, and destination address.	— — — —
2	#1 and #3 take the path R1 → R2 → R4.	— — — —
3	#2 and #4 take the path R1 → R3 → R4. Routers choose the best path of the moment for each packet.	HELL — ORLD —
4	Packets arrive out of order: first #1 and #3, then #4. #2 is lost at a congested router.	HELL — ORLD !
5	TCP notices the missing sequence number; #2 is resent (this time also over the R3 path).	HELL — ORLD !
6	#2 arrives; the receiver arranges the packets by sequence number and reassembles the message.	HELL O, W ORLD !

Table 1.4.8: Trace of the packets of the message "HELLO, WORLD!"



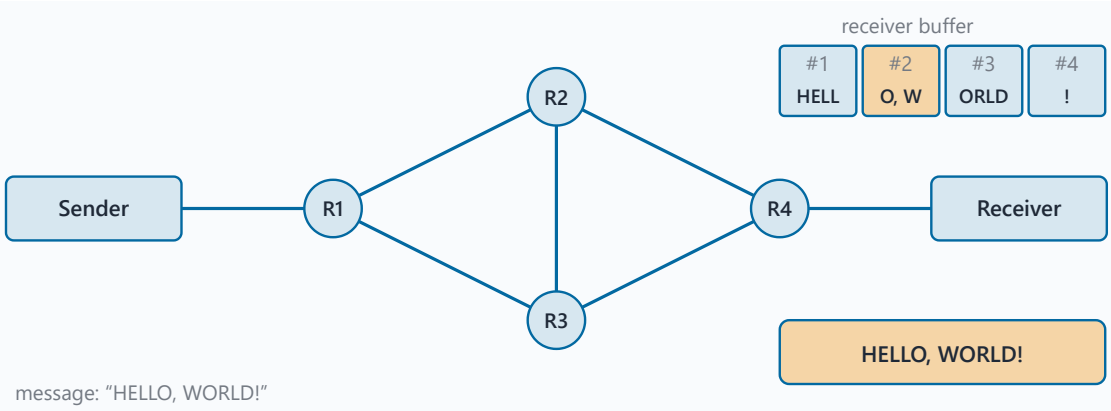


Figure 1.4.14: At the top, step 4: #1, #3, and #4 have arrived and #2 has been lost on the way. At the bottom, the final step: the resent #2 has arrived too, and the message is reassembled as "HELLO, WORLD!"

Three conclusions from the example matter. First, the packets of the same message can take different paths. Second, packets do not have to arrive in the order they were sent: #4 arrived before #2. Third, loss is not an exception but a normal condition of the network: when the buffer of a busy router fills up, newly arriving packets are dropped. Despite all this disorder, the application receives the message complete and in the right order, because TCP takes on the work of ordering and resending.

Common mistake

Thinking that packets flow along a single path and in order, like water in a pipe. IP routes each packet separately; the path and the arrival order can change from packet to packet. In-order delivery is a service provided not by IP but by TCP.

1.4.2.4 A packet crosses the routers

A packet passes through many routers before it reaches its destination. Each router hands the packet to the next router; this single step is called a **hop**. No router knows the whole path. Each one answers only the question "which neighbor should I give a packet bound for this destination?" The answer is written in the router's **routing table**.

Definition 1.4.8

A **routing table** is a table that holds, in each row, an address block and the next hop to which packets for that block are sent. The address block is written in the form `203.0.113.0/24`: the number after the slash shows how many bits from the start of the address define the block. These bits are called the **prefix**. The row `0.0.0.0/0` has a prefix length of zero, so it covers every address; it is the **default route**.

What does a prefix mean? An IPv4 address is 32 bits (1.4.3.1). `/24` says that the first 24 bits of the address, that is, the first three numbers, are fixed. The block `203.0.113.0/24` covers 2^8 = 256 addresses, from 203.0.113.0 to 203.0.113.255. `203.0.0.0/16` fixes only the first two numbers and covers 2^{16} = 65,536 addresses. The longer the prefix, the smaller the block, that is, the more *specific* it is.

Definition 1.4.9

Longest prefix match: If more than one row covers the destination address, the router chooses the row with the longest prefix, that is, the most specific row.

The algorithm every router follows when it receives a packet is short:

```
receive packet, look at destination IP
TTL = TTL - 1;  if 0, discard and notify the source
find the table rows that cover the destination
pick the row with the longest prefix (most specific)
send the packet to that row's next hop
```

The **TTL** (time to live) in the second line is a counter in the IP header. The sender sets it to a starting value (64, for example); each router decrements the TTL by 1 before forwarding the packet. If the TTL falls to zero, the packet is dropped and a "time exceeded" message returns to the sender. This rule prevents a packet that has fallen into a loop between misconfigured routers from circulating in the network forever.

Let us follow an example. A laptop at home (192.168.1.10) sends a packet to a server with the address **203.0.113.10**. There are four routers on the path: R0 at home, the ISP's R1, R2, and R3, which is attached to the server's network. R1 and R2 have other neighbors too (R4, R5, R6), but they are not used for this packet.

Router	Table rows (prefix → next)	Rows covering the destination	Chosen
R0 (home)	192.168.1.0/24 → local network 0.0.0.0/0 → R1 (ISP)	only /0	0.0.0.0/0 → R1
R1 (ISP)	203.0.113.0/24 → R2 198.51.100.0/24 → R4 0.0.0.0/0 → R5	/24 and /0	203.0.113.0/24 → R2
R2	203.0.0.0/16 → R6 203.0.113.0/24 → R3	/16 and /24	203.0.113.0/24 → R3
R3	203.0.113.0/24 → directly connected	/24	delivered to the server

Table 1.4.9: The routers' tables and the choices for destination 203.0.113.10.

Step	Location	TTL	Chosen row	Next
1	laptop	64	—	R0
2	R0	63	0.0.0.0/0	R1 (ISP)
3	R1	62	203.0.113.0/24	R2
4	R2	61	203.0.113.0/24	R3
5	R3	60	203.0.113.0/24	directly connected
6	server	60	—	—

Table 1.4.10: Trace of the packet's journey.

Let us explain the steps one by one. The laptop sees that the destination is not on its own local network (192.168.1.0/24) and gives the packet to the **default gateway**, the home router R0. R0 lowers the TTL to 63. The destination matches only the default route in R0's table; the packet goes to the ISP's R1. If NAT is in use, the packet's source address also changes here (1.4.3.2).

At R1 the TTL becomes 62 and two rows cover the destination: 203.0.113.0/24 and 0.0.0.0/0. The longest prefix wins; /24 is more specific and the packet goes to R2. The R4 and R5 paths are not used. At R2 the TTL becomes 61. Here too two rows cover the destination: 203.0.0.0/16 and 203.0.113.0/24. Again /24 is longer and the packet goes to R3. Had /16 been chosen, the packet would have gone the wrong way, to R6. At R3 the TTL is 60; the destination network is directly connected to R3 and the packet is delivered to the server.

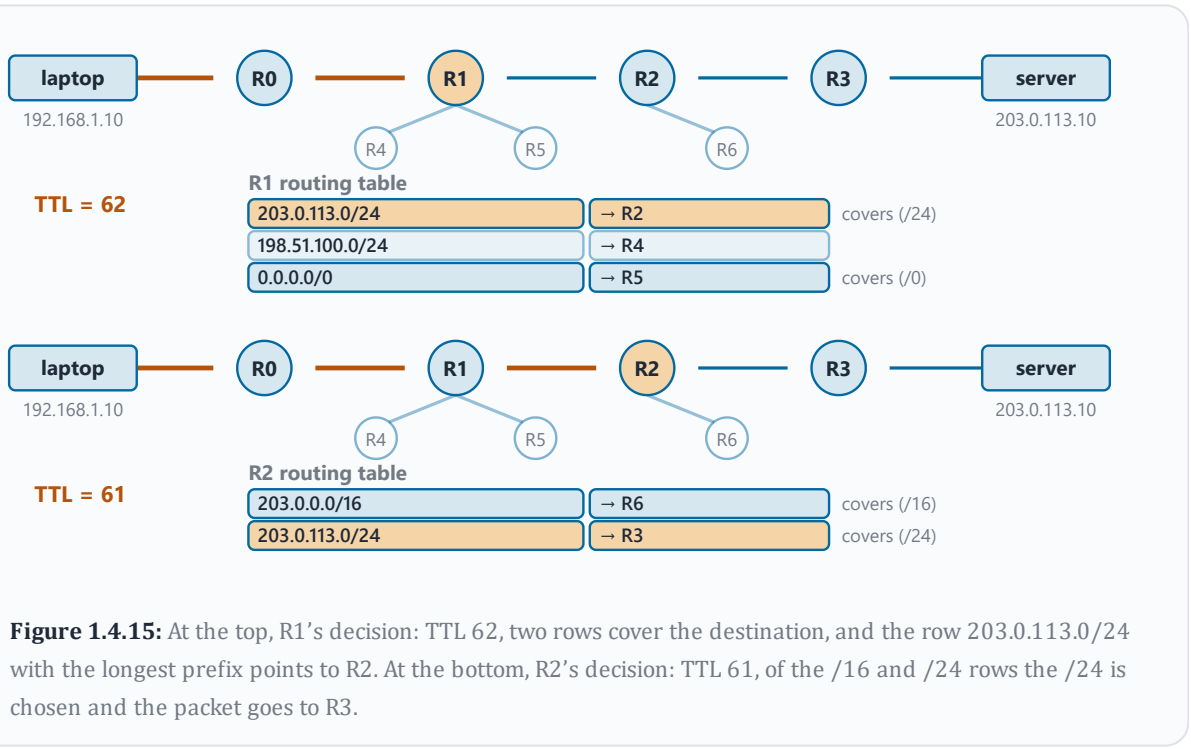


Figure 1.4.15: At the top, R1's decision: TTL 62, two rows cover the destination, and the row 203.0.113.0/24 with the longest prefix points to R2. At the bottom, R2's decision: TTL 61, of the /16 and /24 rows the /24 is chosen and the packet goes to R3.

Example 1.4.5

Let us show at the bit level that the address 203.0.113.10 falls into the blocks 203.0.0.0/16 and 203.0.113.0/24.

1. Convert the address to binary (the method is in 1.4.3.1): 203 = 11001011, 0 = 00000000, 113 = 01110001, 10 = 00001010. The address: `11001011.00000000.01110001.00001010`.
2. The first 16 bits of the block 203.0.0.0/16: `11001011.00000000`. The first 16 bits of the address are the same: the address is in this block.
3. The first 24 bits of the block 203.0.113.0/24: `11001011.00000000.01110001`. The first 24 bits of the address are the same too: the address is in this block as well.
4. There are two matches; the 24-bit match is longer than the 16-bit match. The chosen row is 203.0.113.0/24.

If we make the same comparison for the address 203.0.200.5, the third number is 200 = 11001000, so the /24 block does not match; only the /16 matches, and R2 sends this packet to R6.

traceroute. A by-product of the TTL mechanism is a tool that lets you discover the routers along a path. On Windows it runs as `tracert`, on Linux and macOS as `traceroute`. The tool first sends a packet with a TTL of 1: the first router lowers the TTL to zero, drops the packet, and returns a "time exceeded" message. The address the message comes from is the first router on the path. Then the TTL is raised to 2, 3, and so on, and the routers on the path are listed in order. Each line also shows the round-trip time to that router.

Common mistake

Thinking that a routing table is read from top to bottom and the first matching row is chosen. In a router the order of the rows does not matter: among all the rows that cover the destination, the one with the longest prefix is chosen. (Firewall rules are different: there the first matching rule applies; see 1.4.7.9.)

1.4.3 Addressing: IP, NAT, DNS, URLs, and ports

For a packet to reach the right place, the destination must have an exact address. On the internet this is done with a layered addressing system. The IP address says which device to go to, and the port number says which program on that device to go to. Because people cannot remember addresses made of numbers, domain names are used; DNS translates these names into IP addresses. The URL gathers all this information, together with the path of the requested resource, in one line. This section takes up these addresses in turn.

1.4.3.1 The IP address

Definition 1.4.10

An **IP address** is a numeric address that identifies a network interface and allows packets to be routed to their destination. An **IPv4** address is 32 bits and is written as four numbers separated by dots; each number is between 0 and 255. This way of writing is called **dotted decimal notation**, and each number is called an **octet**. An **IPv6** address is 128 bits and is written as hexadecimal groups separated by colons.

It is deliberate that the definition says "network interface" rather than "device." A laptop with both a wired and a wireless network card can have two separate IP addresses. A router likewise has a separate address for each network it is attached to.

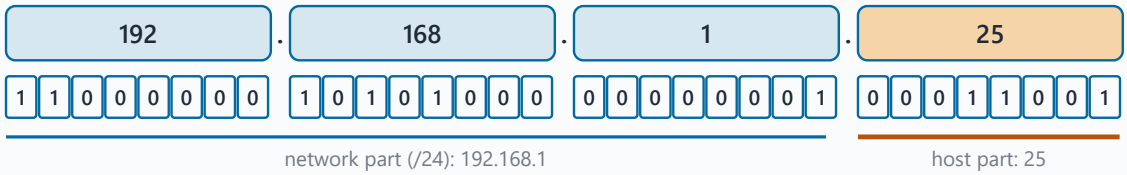


Figure 1.4.16: The four octets of the address 192.168.1.25 and the 8-bit binary form of each. In a /24 network the first three octets identify the network (192.168.1) and the last octet identifies the host (25).

Why 0–255? Each octet is 8 bits. With 8 bits you can write $2^8 = 256$ different values: the smallest is 00000000 = 0 and the largest is 11111111 = 255. Four octets make $4 \times 8 = 32$ bits in total and give $2^{32} = 4{,}294{,}967{,}296$ different addresses. At first glance this number looks large, but the world's population has passed 8 billion, and there is not even one address per person. On top of that, one person's phone, laptop, television, and smartwatch each need separate addresses. IPv4 addresses have therefore run out; NAT (1.4.3.2) and IPv6 are two solutions to this problem.

Converting decimal to binary. The easiest way to convert an octet to binary is to try the bit weights from largest to smallest. The weights of the eight bits, from left to right, are 128, 64, 32, 16, 8, 4, 2, 1. At each step, if the number is greater than or equal to the weight, write 1 for that bit and subtract the weight from the number; otherwise write 0 and leave the number unchanged.

Example 1.4.6

Let us convert the octets 168 and 25 in the address 192.168.1.25 to binary.

Weight	For 168: number ≥ weight?	Bit	Remainder	For 25: number ≥ weight?	Bit	Remainder
128	168 ≥ 128 yes	1	40	25 ≥ 128 no	0	25
64	40 ≥ 64 no	0	40	25 ≥ 64 no	0	25

Weight	For 168: number ≥ weight?	Bit	Remainder	For 25: number ≥ weight?	Bit	Remainder
32	40 ≥ 32 yes	1	8	25 ≥ 32 no	0	25
16	8 ≥ 16 no	0	8	25 ≥ 16 yes	1	9
8	8 ≥ 8 yes	1	0	9 ≥ 8 yes	1	1
4	0 ≥ 4 no	0	0	1 ≥ 4 no	0	1
2	0 ≥ 2 no	0	0	1 ≥ 2 no	0	1
1	0 ≥ 1 no	0	0	1 ≥ 1 yes	1	0

168 = `10101000` (check: 128 + 32 + 8 = 168) and 25 = `00011001` (check: 16 + 8 + 1 = 25). Since 192 = 128 + 64, it is `11000000`, and 1 is `00000001`. The whole address: `11000000.10101000.00000001.00011001`.

Every octet is written with exactly 8 bits; leading zeros are not dropped. The binary form of 25 is written "00011001," not "11001." Otherwise it becomes unclear which bit of the 32-bit address belongs to which octet, and prefix comparison comes out wrong.

IPv6. A 128-bit IPv6 address can take $2^{128} \approx 3.4 \times 10^{38}$ different values; that is about 7.9×10^{28} times the number of IPv4 addresses. The address is divided into eight groups; each group is 16 bits and is written with four hexadecimal digits (each hexadecimal digit is 4 bits). There are two rules for shortening long addresses: leading zeros of a group may be omitted, and consecutive all-zero groups may be shown once with `::`.

Example 1.4.7

What is the full form of the address `2001:db8::25`?

1. There are three visible groups: `2001`, `db8`, and `25`. The total number of groups must be eight; `::` stands for the missing 8 − 3 = 5 zero groups.
2. Each group is padded to four digits: `db8` → `0db8`, `25` → `0025`.
3. Full form: `2001:0db8:0000:0000:0000:0000:0000:0025`.

`::` can be used only once in an address; if it were used twice, it would be unclear how many zero groups lay on each side.

Common mistake

Writing 256 or a larger number in an octet. 192.168.1.256 is not a valid IPv4 address; 8 bits can count only up to 255. Likewise, letters such as "g" or "z" cannot appear in an IPv6 address: hexadecimal digits are only 0–9 and a–f.

1.4.3.2 Local and public IP addresses: NAT

If you look at the network settings of a laptop at home, you will often see an address like 192.168.1.10. But if you visit a website that asks “what is my IP address?” from the same laptop, a completely different address appears, for example 203.0.113.7. The two addresses being different is not an error; the router performs the conversion between them.

Definition 1.4.11

A **private IP address** is an address that is valid only inside a local network and is not routed on the internet. Three blocks are reserved for this: 10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16. A **public IP address** is an address that is unique on the internet and can be reached directly. **NAT** (network address translation) is the router replacing local addresses with its own public address. The common form that also changes port numbers is called **NAPT**.

The private address blocks are reused in every home and every school. A device with the address 192.168.1.10 can exist in a home in Giresun and in a home in Istanbul alike. These addresses are unique inside the network but not unique in the world. That is why the source address of every packet leaving for the internet must be replaced with the router’s public address.

The home in the example has three devices: a laptop (192.168.1.10), a phone (192.168.1.11), and a tablet (192.168.1.12). The router’s internal address is 192.168.1.1 and its external (public) address is 203.0.113.7. The laptop connects to a server with the address 198.51.100.20.

Step	Location	Source	Destination	NAT table
1	Home network (outbound)	192.168.1.10:50000	198.51.100.20	empty
2	Leaving the router	203.0.113.7:61000	198.51.100.20	192.168.1.10:50000 ↔ 203.0.113.7:61000
3	Internet (return)	198.51.100.20	203.0.113.7:61000	same
4	Home network (return)	198.51.100.20	192.168.1.10:50000	same

Table 1.4.11: Change of addresses with NAT (laptop 192.168.1.10, public address 203.0.113.7, server 198.51.100.20).

1. The laptop sends the packet. The source address is the private IP (192.168.1.10), and this private address is not routed on the internet.
2. The router replaces the source address with its own public address (203.0.113.7) and the source port with a port of its own choosing (61000). To be able to find out later whom the reply belongs to, it writes this mapping into the **NAT table**.
3. The server sees only the address 203.0.113.7 and sends the reply to this address, to port 61000. All the devices at home look like a single address from the outside.

4. The router looks at the destination port of the reply (61000); according to the NAT table, this connection belongs to the laptop. It changes the destination to 192.168.1.10:50000 and forwards the packet.

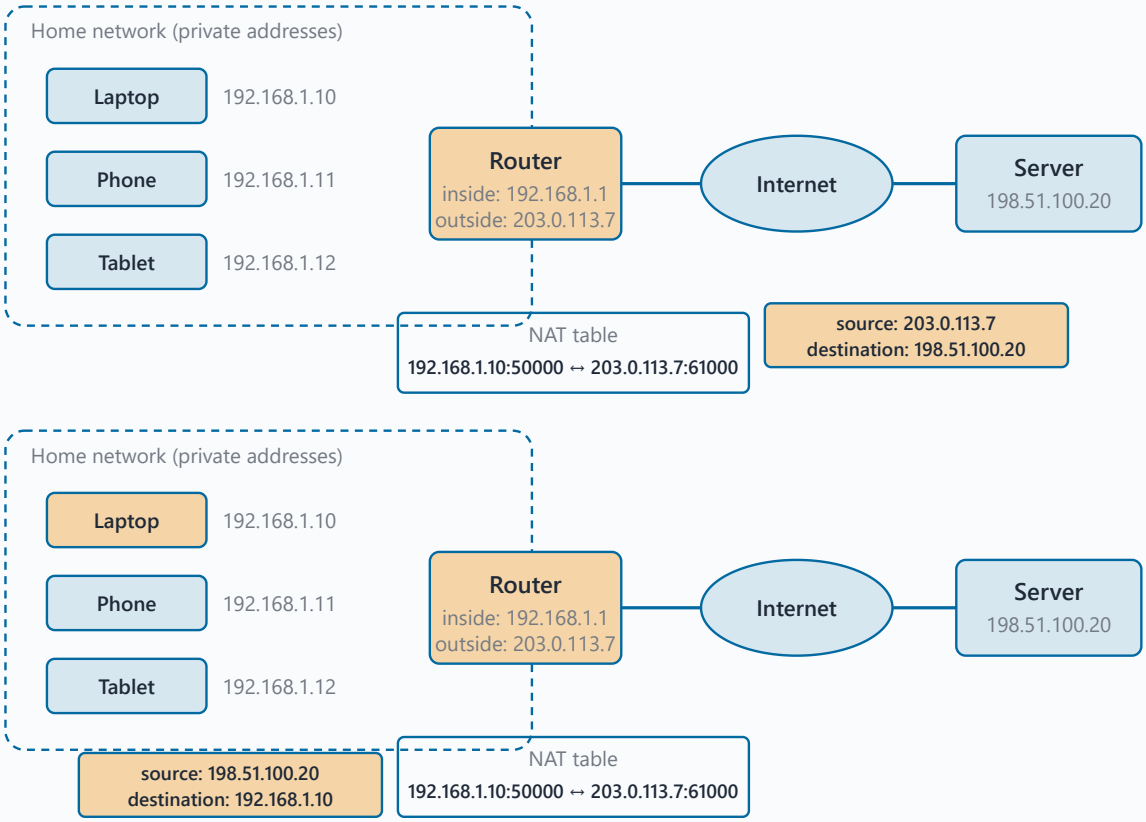


Figure 1.4.17: At the top, outbound: the router sets the source to 203.0.113.7 and writes the mapping into the NAT table. At the bottom, return: using the mapping in the table, the destination is translated to 192.168.1.10.

Why is the port number changed? At home, the phone and the laptop may connect to the same server at the same time, and by chance they may have picked the same source port. If the router gives each connection a different outside port, it can tell the returning replies apart without mixing them up. A port number is 16 bits and takes 2^{16} = 65,536 values; so tens of thousands of connections can run at once behind a single public address.

Example 1.4.8

Which of these addresses are private (local) addresses? 10.5.5.5 · 172.31.255.1 · 172.32.0.1 · 203.0.113.7 · 192.168.1.10.

1. 10.5.5.5: the first octet is 10; it is in the 10.0.0.0/8 block. **Private.**
2. The 172.16.0.0/12 block extends from 172.16.0.0 to 172.31.255.255 (/12 fixes the whole first octet and the first 4 bits of the second octet; the second octet can vary from 16 to 31). 172.31.255.1 is in this range: **private.**

3. The second octet of 172.32.0.1 is 32, which is outside the range: **public.**
4. 203.0.113.7 is in none of the three blocks: **not private.**
5. 192.168.1.10 is in the 192.168.0.0/16 block: **private.**

Note: the blocks 203.0.113.0/24 and 198.51.100.0/24 are reserved for use in documentation and examples; they are never assigned to real devices. That is why the example addresses in these notes are drawn from these blocks.

Common mistake

Thinking that the 192.168.x.x address shown in a computer's network settings is the address visible on the internet. This address is valid only in the home network. Websites see the router's public address; everyone in the same home goes out to the internet with the same public address.

1.4.3.3 Domain names and DNS

You can connect to a website by a number such as 203.0.113.10, but such numbers are hard to remember. Moreover, when an organization changes its server, the IP address may change too. To solve these two problems, human-readable names are used.

Definition 1.4.12

A **domain name** is a human-readable name, for example `www.ornek.edu.tr`. **DNS** (Domain Name System) is a worldwide distributed name database that translates a domain name into the IP address to which a connection can be made. This translation process is called **DNS resolution**.

> `www.ornek.edu.tr` → DNS query → `203.0.113.10` → web server

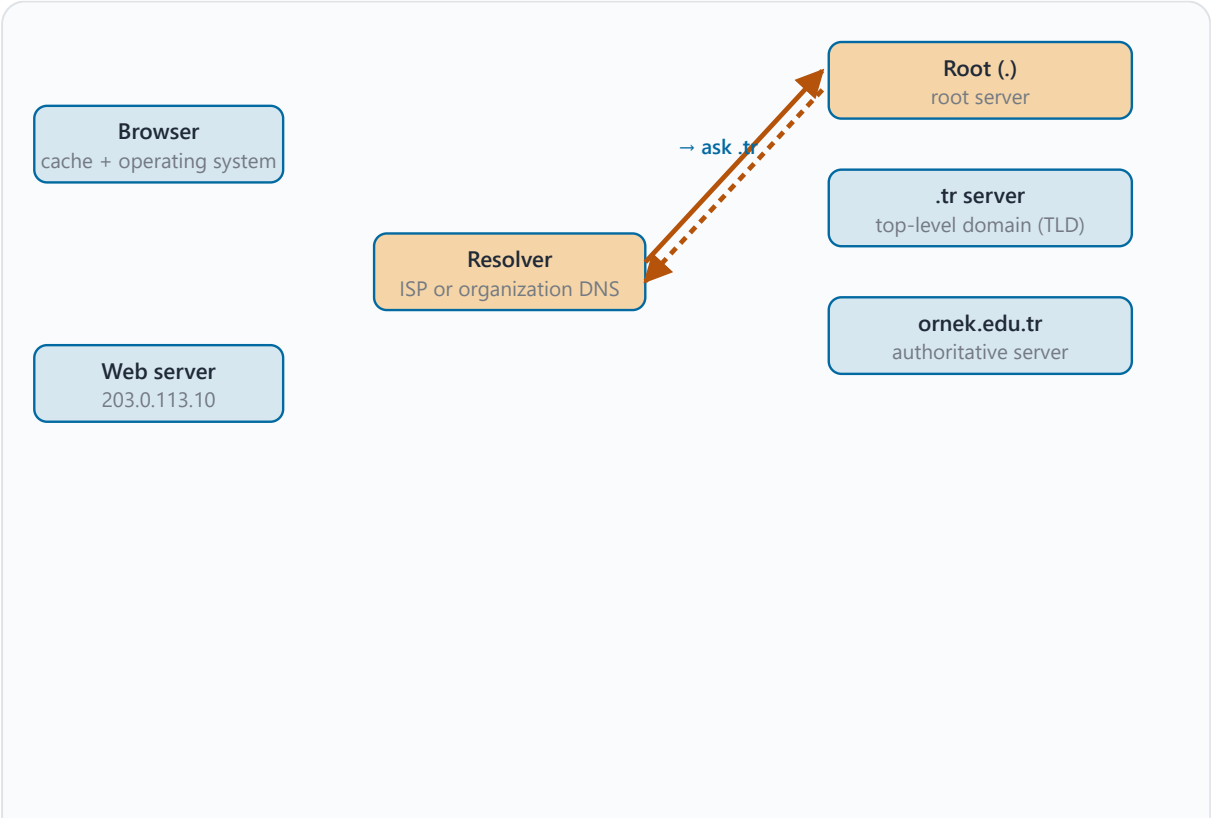
DNS is commonly compared to a phone book: you know the person's name, and the book gives you the number. But DNS is not a single phone book. Keeping all the domain names in the world on one server would be an enormous load, and if that server went down the whole internet would stop. DNS is therefore built to be hierarchical and distributed: each server is responsible for only a part of the name.

1.4.3.4 DNS resolution step by step

Four kinds of server take part in resolution. The **DNS resolver** is the server that asks all the questions on the user's behalf and brings back the result; it is usually the ISP's or the organization's server. The **root server** is at the very top of the hierarchy and knows the servers of the top-level domains (.tr, .com, .org, ...). The **top-level domain server** (the .tr server, for example) knows the authoritative servers of the domains beneath it. The **authoritative server** holds the records of a particular domain and gives the final answer.

Suppose a user opens `www.ornek.edu.tr` for the first time and no cache holds any record.

1. The user types `www.ornek.edu.tr` in the address bar. To connect, the IP address of this name is needed.
2. The browser and the operating system first check their own caches: there is no record.
3. The question is sent to the DNS resolver in the network settings.
4. The resolver starts from the very top. The root server does not know the name, but says "ask the .tr servers."
5. The .tr server does not know the full address either: it says "ask the authoritative server of ornek.edu.tr."
6. The authoritative server knows the record and answers: `203.0.113.10`.
7. The resolver passes the answer to the browser and keeps the record in its cache for a certain time; if the same question comes again, it answers immediately.
8. Now that the browser knows the IP address, it connects to the web server at 203.0.113.10.



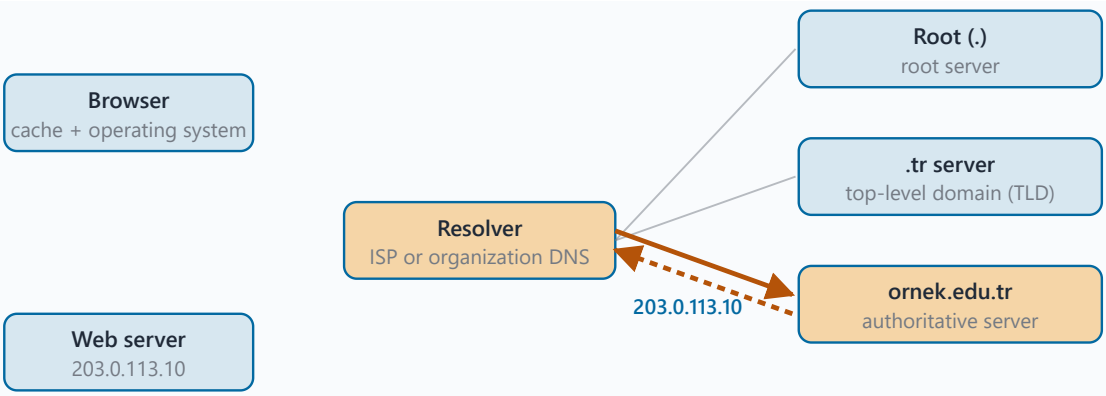


Figure 1.4.18: Two steps of DNS resolution. At the top, the resolver asks the root server and receives the referral "ask .tr." At the bottom, the authoritative server knows the record and returns the answer 203.0.113.10.

Notice that the root and .tr servers do not give the real answer; they only say whom to ask next. These answers are called **referrals**. The resolver descends the name hierarchy from top to bottom, reaching a somewhat more specific server at each step. The user's computer sees none of the steps of this journey: it asks one question and gets one answer.

1.4.3.5 The resolver's cache

Starting at the root server and asking three separate servers for every name would be slow and a heavy load on the root servers. Resolvers therefore keep everything they learn in a **cache** for a while. Not only the final answer but also the intermediate referrals are stored: "this is the .tr server," "this is the authoritative server of ornek.edu.tr." How long each record is kept is set by the authoritative server that issued it, with a **TTL** value.

The algorithm the resolver follows when a question arrives is this:

```
is the name in the cache?  Yes → answer, done
server = closest known (else root)
ask the server for the name
referral: write it to the cache, ask that server
if an IP arrives, cache it (TTL), answer
```

The second line is important: the resolver does not always start from the root; it starts from the closest server it knows in its cache. Table 1.4.12 follows three questions arriving one after another at a newly started resolver with an empty cache.

Question	What was found in the cache?	Servers asked	Added to the cache	Answer	Queries
`www.ornek.edu.tr`	nothing	root → .tr → authoritative	.tr → server; ornek.edu.tr → authoritative; www → 203.0.113.10 (TTL 1 hour)	203.0.113.10	3
`student.ornek.edu.tr` (same minute)	no entry for the name, but the authoritative server is known	authoritative	student → 203.0.113.20	203.0.113.20	1
`www.ornek.edu.tr` (another user)	entry found	—	—	203.0.113.10	0

Table 1.4.12: Trace of the resolver cache. The last column is the number of queries sent to external servers for that question.

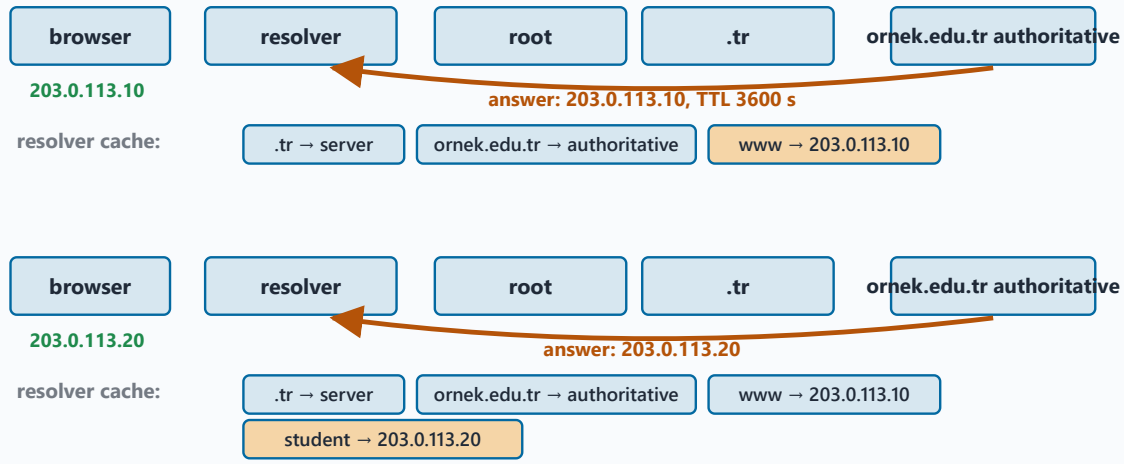


Figure 1.4.19: At the top, the end of the first question: the authoritative server gives the answer 203.0.113.10 with a TTL of 3600 s; the cache holds three records. At the bottom, the second question: the root and .tr are skipped, the authoritative server is asked directly, and the student record is added to the cache.

The first question needs three queries. The second ends with a single query, because the authoritative server of ornek.edu.tr is now known; the root and .tr steps are skipped. The third question needs no external query at all and the answer returns immediately. Because real resolvers serve millions of users, most popular names are always found in the cache.

The cache has a side effect, too. A TTL of 3600 s is 1 hour; the record is not deleted before this time is up. If the organization moves its server in the meantime and changes the IP address, resolvers that hold the old record in their cache keep giving the old address for up to an hour. This is why an address change does not reach everyone at once. Organizations shorten the TTL temporarily before a planned move.

Common mistake

Thinking that the TTL in DNS and the TTL in the IP header are the same thing. The TTL in an IP packet is a hop counter and drops by 1 at each router (1.4.2.4). The TTL in a DNS record is a duration in seconds and says how long the record is kept in the cache. They have the same name because both mean "time to live," but what they measure is different.

1.4.3.6 The parts of a domain name

A domain name consists of labels separated by dots and is read **from right to left**, from general to specific. Take the name `www.giresun.edu.tr`:

- **tr:** the country-code **top-level domain** (TLD). It denotes Turkey; .de denotes Germany and .jp denotes Japan. There are also generic top-level domains not tied to a country, such as .com and .org.
- **edu:** the type of organization. In Turkey, .edu.tr is reserved for educational institutions, .gov.tr for public bodies, and .com.tr for companies.
- **giresun:** the organization's domain name. The organization has registered this name and manages the names beneath it itself.
- **www:** the **subdomain** or server name. Under the same domain the organization can open other subdomains too, such as obs, mail, or library.

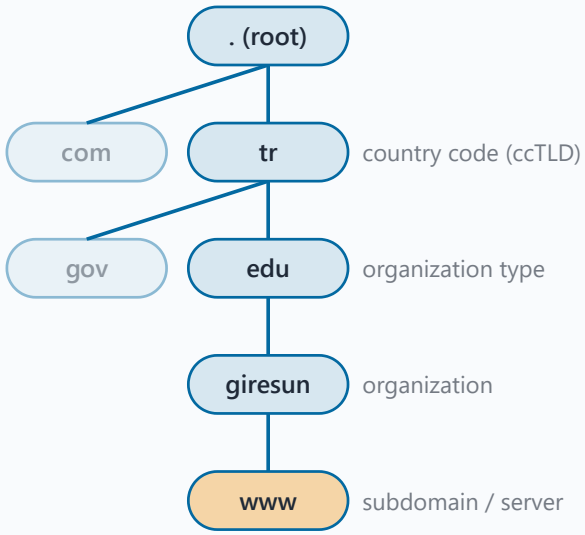


Figure 1.4.20: The DNS name tree. Under the root (.) are the top-level domains (tr, com), under them the organization types (edu, gov), further down the organization (giresun), and at the bottom the server name (www).

A DNS query follows this tree from the root down: first the root, then tr, then edu.tr and giresun.edu.tr. Because the name is read from right to left, resolution proceeds from right to left. This arrangement also distributes responsibility. The root servers know only the top-level domains, and the .tr servers know only the domains under .tr. The names inside giresun.edu.tr are held only by the university's authoritative server.

Common mistake

Reading the domain name from left to right. The address `giresun.edu.tr.example.com` belongs not to Giresun University but to the owner of the `example.com` domain; the name is read from right to left, and the part that identifies the organization is the registered domain at the far right. Phishing attacks exploit this misreading (1.4.7.4).

1.4.3.7 The structure of a URL

Definition 1.4.13

A **URL** (Uniform Resource Locator) is the complete address of a resource on the internet. In one line it says with which protocol, to which server to connect, and which resource is requested on that server.

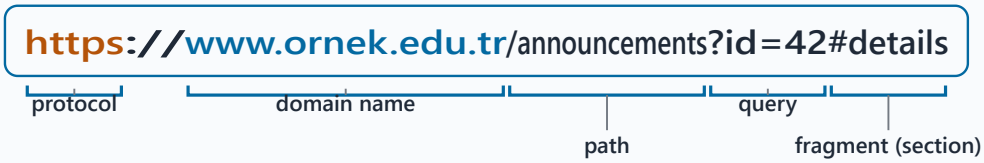


Figure 1.4.21: The parts of the address `https://www.ornek.edu.tr/announcements?id=42#details` : protocol, domain name, path, query, and fragment.

Part	In the example	Meaning
Protocol	`https`	Which rules are used to communicate (encrypted HTTP).
Domain name	`www.ornek.edu.tr`	The name of the server; DNS turns it into an IP address.
Path	`/announcements`	The path of the resource on the server.
Queries	`?id=42`	Extra information sent to the server, in the form `name=value` ; several pairs are separated by `&` .
Fragment	`#details`	A section within the page; it is not sent to the server, and the browser scrolls to that section.

Table 1.4.13: The parts of a URL.

A URL can have one more part besides these: the **port**. It is written after the domain name with a colon, for example `http://203.0.113.10:8080`. If no port is written, the protocol's **default port** is used: 80 for HTTP, 443 for HTTPS. Note also that in this example an IP address is written directly instead of a domain name; in that case DNS is not consulted at all.

Example 1.4.9

Let us break the address `http://203.0.113.10:8080/report?year=2025&month=3#summary` into its parts. Which part is not sent to the server? Which port does it connect to?

1. Protocol: `http`. It is unencrypted.
2. Server: `203.0.113.10`. It is an IP address; no DNS query is needed.
3. Port: `8080`. Because it is written explicitly, HTTP's default port 80 is not used.
4. Path: `/report`.
5. Query: `?year=2025&month=3`. There are two `name=value` pairs: year = 2025 and month = 3.
6. Fragment: `#summary`. It is not sent to the server; after the page loads, the browser scrolls to the section with the id "summary."

The request sent to the server contains only the `/report?year=2025&month=3` part and is sent to port 8080 of the address 203.0.113.10.

Common mistake

Thinking that the fragment (`#…`) is sent to the server, or confusing the query (`?…`) with the fragment. The query can change the server's response; the fragment is processed only in the browser. Personal or confidential information must not be put in any part of a URL, because URLs are openly visible in browser history, in server logs, and in shared links.

1.4.3.8 Ports

Many services can run on a server at the same time: a web server, an email server, a remote login service, a DNS service. The IP address says which computer a packet goes to, but not which service on that computer. The port number does this.

Definition 1.4.14

A **port** is a 16-bit number that identifies a particular service or connection endpoint on a device (0–65,535). An IP address is like a building, and a port number like an apartment in that building. Commonly used ports: **80** HTTP, **443** HTTPS, **22** SSH (encrypted remote login), **53** DNS.

1. The browser opens `https://ornek.edu.tr`. DNS supplies the IP address; the default port for HTTPS is 443. The packet's destination is written as IP:port, that is, 203.0.113.10:443.
2. The packet first reaches the IP address, that is, the building, by way of the routers. IP decides only which computer to go to.
3. The server's operating system looks at the port number, that is, the apartment. It delivers the packet to the service listening on that port, here the HTTPS web server. A packet arriving at port 80 goes to one service, and a packet arriving at port 22 goes to yet another.
4. If a port is written in the URL, it is used instead of the default port: a packet sent to `http://203.0.113.10:8080` reaches a different application listening on 8080 (a test server, for example).

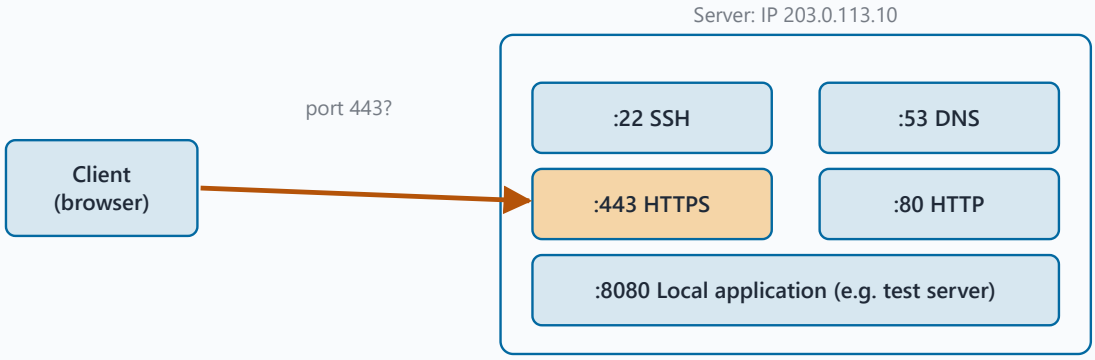


Figure 1.4.22: On the server at 203.0.113.10, five services run on different ports. A packet arriving at port 443 is delivered to the HTTPS web server.

There is also a port on the client side of the connection. When the browser opens a connection, the operating system gives it a temporary port (50000, for example). The server's reply comes to this port. This way, ten tabs open on the same computer can each connect separately to the same server: each tab's connection uses a different client port. The port rewriting done by NAT (1.4.3.2) is done on these client ports.

Common mistake

Confusing a network port with the physical sockets on the back of a computer (USB, HDMI, or a switch's cable socket). A network port is not a physical hole but a number the operating system uses to tell connections apart.

1.4.4 The TCP connection and encrypted communication

IP carries packets only on a "best-effort" basis: a packet may be lost, reordered, or duplicated. A web page or a bank transaction, however, must arrive complete and in the right order. The content must also stay secret from anyone who could read the data along the way. This section describes how these two needs are met: first TCP's reliable connection, then HTTPS's encryption.

1.4.4.1 TCP: three-way handshake, data, and closing

Before sending data between two ends, TCP sets up a **connection**. While the connection is being set up, the two sides tell each other their initial sequence numbers. As data is transferred, every byte gets a number, and the receiver reports up to which byte it has received with an **acknowledgment** (ACK). When the job is done, the connection is closed in an orderly way in both directions.

Definition 1.4.15

The **three-way handshake** is the setup of a TCP connection with three segments: the client sends **SYN** (synchronize), the server replies with **SYN + ACK**, and the client confirms with **ACK**. The **sequence number** (seq) is the number of the first byte in a segment. The **acknowledgment number** (ack) means "I have received everything up to this number; this is the next byte I expect."

In the example, the client connects to a server listening on port 443. The whole process in brief:

```
client → SYN
server → SYN + ACK
client → ACK        // connection established
client → data (HTTP request)
server → ACK
client → FIN;  server → ACK
server → FIN;  client → ACK  // closed
```

Let us follow it with numbers. Let the client's initial sequence number be 1000 and the server's 5000. In reality these numbers are chosen at random for each connection; this way, segments left over from an old connection do not get mixed up with the new one, and it is harder to inject a forged segment

from outside.

#	Direction	Segment	seq	ack	Client state	Server state
1	client → server	SYN	1000	—	SYN-SENT	SYN-RECEIVED
2	server → client	SYN + ACK	5000	1001	SYN-SENT	SYN-RECEIVED
3	client → server	ACK	1001	5001	ESTABLISHED	ESTABLISHED
4	client → server	data (100 bytes)	1001	—	ESTABLISHED	ESTABLISHED
5	server → client	ACK	—	1101	ESTABLISHED	ESTABLISHED
6	client → server	FIN	1101	—	FIN-WAIT	CLOSE-WAIT
7	server → client	ACK	—	1102	FIN-WAIT	CLOSE-WAIT
8	server → client	FIN	5001	—	TIME-WAIT	LAST-ACK
9	client → server	ACK	—	5002	TIME-WAIT	CLOSED

Table 1.4.14: Trace of a TCP connection (client's initial sequence number 1000, server's 5000).

Let us explain the rows in order:

1. **SYN.** The client requests a connection and announces its initial sequence number (1000).
2. **SYN + ACK.** The server accepts: its own sequence number is 5000 and ack = 1001, that is, "I received 1000, I expect 1001." Although a SYN segment carries no data, it uses up one sequence number; that is why the ack is 1001.
3. **ACK.** The client acknowledges the server's SYN with ack = 5001. The connection has been set up with three segments, but no data has been sent yet.
4. **Data.** The client sends a 100-byte HTTP request. The number of the first byte is 1001; the bytes are numbered 1001 through 1100. With HTTPS, a TLS negotiation is also carried out over this channel first (1.4.4.4).
5. **ACK.** The server reports with ack = 1101 that it has received all 100 bytes: 1001 + 100 = 1101. Had no acknowledgment arrived, the client would have waited a while and resent the same bytes.
6. **FIN.** The work is done; the client closes its own direction with FIN.
7. **ACK.** The server acknowledges the FIN with ack = 1102. A FIN, like a SYN, uses up one sequence number. If the server still has data to send, it can still send it; the connection has been closed in only one direction for now.
8. **FIN.** The server closes its own direction too (seq = 5001).
9. **ACK.** The client sends the final acknowledgment (ack = 5002). The connection is now closed in both directions.

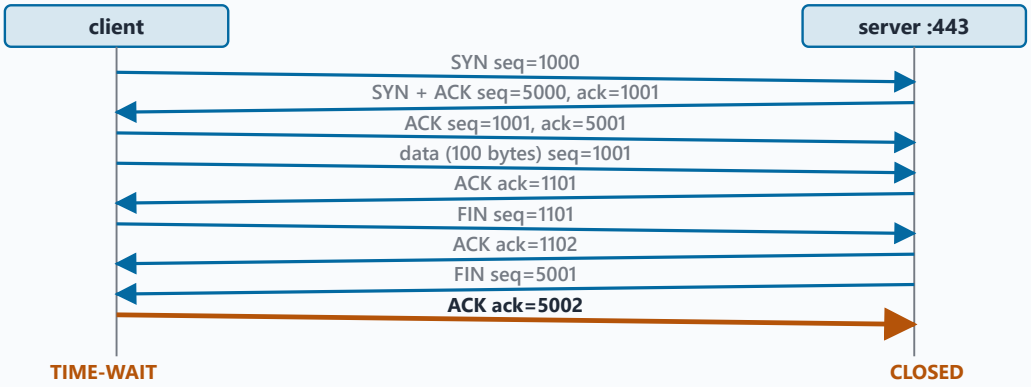


Figure 1.4.23: All the segments of a TCP connection on a timeline. From top to bottom: the three-way handshake, 100 bytes of data and its acknowledgment, and the four-segment closing. The labels at the bottom show the final state of the two sides.

Setting up takes 3 segments and closing takes 4. Closing takes four segments because the two directions are closed independently: when one side says "I have nothing more to say," the other side can keep sending data. UDP has none of these steps; UDP sends the data directly in the first packet without setting up a connection.

Example 1.4.10

On another connection, the client's initial sequence number is 300 and the server's is 9000. The client sends a 250-byte request and the server sends a 1200-byte response. What are the seq and ack values in the segments?

1. SYN: seq = 300.
2. SYN + ACK: seq = 9000, ack = 300 + 1 = 301.
3. ACK: seq = 301, ack = 9000 + 1 = 9001.
4. Request: seq = 301; bytes 301 through 301 + 250 − 1 = 550. The server's acknowledgment is ack = 551.
5. Response: seq = 9001; bytes 9001 through 9001 + 1200 − 1 = 10,200. The client's acknowledgment is ack = 10,201.

The rule is the same every time: acknowledgment number = number of the last byte received + 1. SYN and FIN each use up one number.

Common mistake

Taking the acknowledgment number to be the number of the last byte received. ack = 1101 does not mean "I have received up to 1101"; it means "I have received up to 1100, and the next byte I expect is 1101." Another common mistake is to forget that SYN uses up a sequence

number and write the first data byte as 1000.

1.4.4.2 HTTP and HTTPS

HTTP is the protocol of the conversation between the browser and the web server. The browser says "give me this resource," and the server sends it. This conversation is in plain text: any device along the way can read the packets. HTTPS encrypts the same conversation with **TLS** (Transport Layer Security).

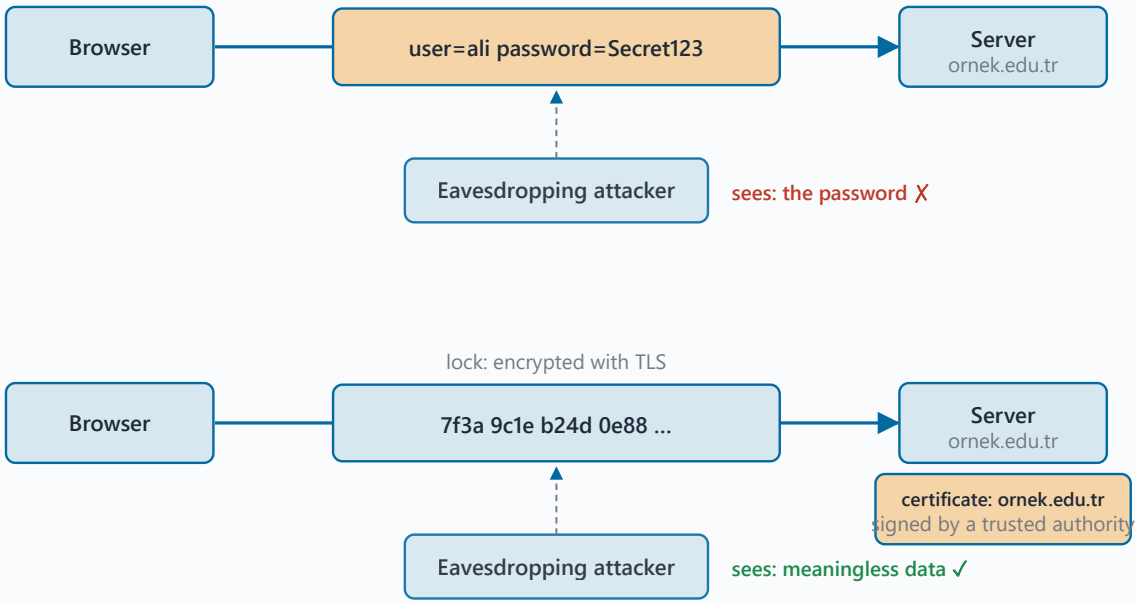


Figure 1.4.24: At the top, HTTP: an eavesdropping attacker on the same network sees the username and password in plain text. At the bottom, HTTPS: the same attacker sees only meaningless bytes; the browser also checks the server's certificate issued for the name ornek.edu.tr.

Let us see the difference in a concrete case. A student logs in to a site over the open Wi-Fi network at a cafe. If the site uses HTTP, someone on the same network can capture the packets and read a line such as `user=ali password=Secret123` in the clear. If the site uses HTTPS, the same packets show meaningless bytes such as `7f3a 9c1e b24d 0e88 …`. The "connection is secure" indicator in the browser's address bar (a lock icon in older versions) shows that the connection is encrypted.

HTTPS's second job is identity verification. Encryption alone is not enough: if the user has unknowingly connected to the attacker's server, the data goes to the attacker even if it is encrypted. So the server presents a **certificate** signed by a trusted organization. The certificate says, "this public key really belongs to the domain name ornek.edu.tr." The browser checks the signature; if the certificate was issued to a different name or has expired, it shows a warning.

Common mistake

Reading the lock icon as “this site is trustworthy.” The lock guarantees two things: the traffic is encrypted, and the certificate was issued to the domain name in the address bar. It does not guarantee that the site is honest. Phishing sites also easily obtain certificates for their own domain names and use HTTPS. The thing to look at is the domain name itself.

1.4.4.3 Encryption basics

Encryption is turning readable text (**plaintext**) into text that looks meaningless, with the help of a key. The reverse process is called **decryption**. Someone who does not know the key cannot read the content even if they see the encrypted text. There are two approaches, depending on the type of key used; to them is added hashing, which is often confused with encryption.

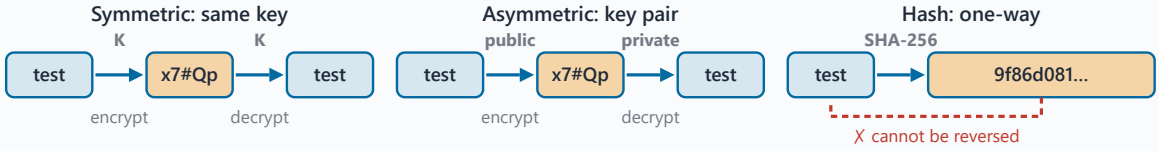


Figure 1.4.25: From left to right: in symmetric encryption the same key (K) encrypts and decrypts; in asymmetric encryption the public key encrypts and the private key decrypts; a hash (SHA-256) is one-way and cannot be reversed.

Definition 1.4.16

Symmetric encryption: the same key is used for encryption and decryption. It is fast; the problem is delivering the key safely to the other side. Example: AES.

Asymmetric encryption: each party has a key pair. The **public key** is given to everyone and is used for encrypting or verifying a signature. The **private key** stays secret and is used for decrypting or signing. It is slower, but it solves the key distribution problem. Examples: RSA, elliptic curves.

Hash: produces a fixed-length “fingerprint” from data. It is not encryption; it cannot be reversed. It is used in storing passwords and checking integrity. Example: SHA-256.

The logic of symmetric encryption. The oldest known example is the Caesar cipher: each letter is shifted a fixed number of places along the alphabet. In the 26-letter English alphabet, with a shift of 3, A → D, B → E, and the word “NET” is written as “QHW.” The receiver knows the same key (3) and decrypts by shifting the letters back by 3. Today this cipher is broken in a few seconds, because there

are only 25 different keys and all of them can be tried. Real algorithms such as AES use 128- or 256-bit keys; trying 2^{128} keys is beyond any computing power one could imagine. But the principle is the same: both sides must hold the same secret key.

Here lies the problem. How will a browser and a server it has never met agree on a shared secret key over the internet? If they send the key in the open, an eavesdropper along the way learns it too. Asymmetric encryption solves this problem.

Example 1.4.11

Let us see how asymmetric encryption works with small numbers, in a toy RSA example. Here *a mod n* is the remainder of dividing a by n.

1. The server chooses two prime numbers: p = 5, q = 11. Their product is n = 55.
2. Compute (p − 1)(q − 1) = 4 · 10 = 40. Choose the public exponent e = 3; find the private exponent d so that e · d leaves remainder 1 when divided by 40: d = 27, because 3 · 27 = 81 = 2 · 40 + 1.
3. The public key (n = 55, e = 3) is announced to everyone. The private key d = 27 stays only on the server.
4. The browser wants to send the number 7. It encrypts with the public key: $7^3 = 343$ and 343 mod 55 = 13 (because 6 · 55 = 330, 343 − 330 = 13). The number 13 goes out on the wire.
5. The server decrypts with the private key: 13^{27} mod 55 = 7. The original number is back.

An eavesdropper sees 13 and the public key (55, 3); to find d, they must factor 55 into primes. 55 = 5 · 11 is found immediately. In real RSA keys, however, n has hundreds of digits (2048 bits). There is no known fast way to factor such a number. Security rests on this difficulty.

Hash functions. SHA-256 produces a 256-bit digest (64 hexadecimal digits) however long the input is. The SHA-256 hash of the word `test` begins with `9f86d081884c7d65…`. The smallest change in the input changes the hash completely: `Test` (with a capital T) gives `532eaabd9574880d…`, and `test␣` with a space added at the end gives `231aa78c52a74a57…`. There is no way to compute the input back from the hash.

These properties serve two purposes. **Integrity checking:** when downloading a file, the hash published by the site is compared with the hash of the downloaded file; if even a single bit has changed, the hashes do not match. **Password storage:** well-designed systems store not the user's password but the hash of the password. At login, the hash of the typed password is computed and compared with the stored one. Even if the database is stolen, the passwords cannot be read directly.

Common mistake

Saying “passwords are encrypted with SHA-256.” A hash is not encryption: it uses no key and cannot be reversed. “Encrypted” data can be opened again with the right key; “hashed” data cannot be opened again with any key. The two concepts serve different purposes.

1.4.4.4 How the methods work together in HTTPS

Symmetric encryption is fast but has a key distribution problem. Asymmetric encryption solves key distribution but is slow. HTTPS combines the strengths of both: it uses the asymmetric method only once, to agree on a shared session key, and then encrypts the actual data symmetrically with that key.

1. The server sends the browser its certificate and public key. The private key stays secret on the server. The browser checks the certificate.
2. The browser generates a random **session key** (K), encrypts it with the server’s public key, and sends it. This is the asymmetric step; an eavesdropper along the way cannot decrypt K.
3. The server decrypts K with its own private key. Now both sides hold the same secret key, and asymmetric encryption has finished its job.
4. The actual data (page, form, password) is encrypted symmetrically with K and flows in both directions.

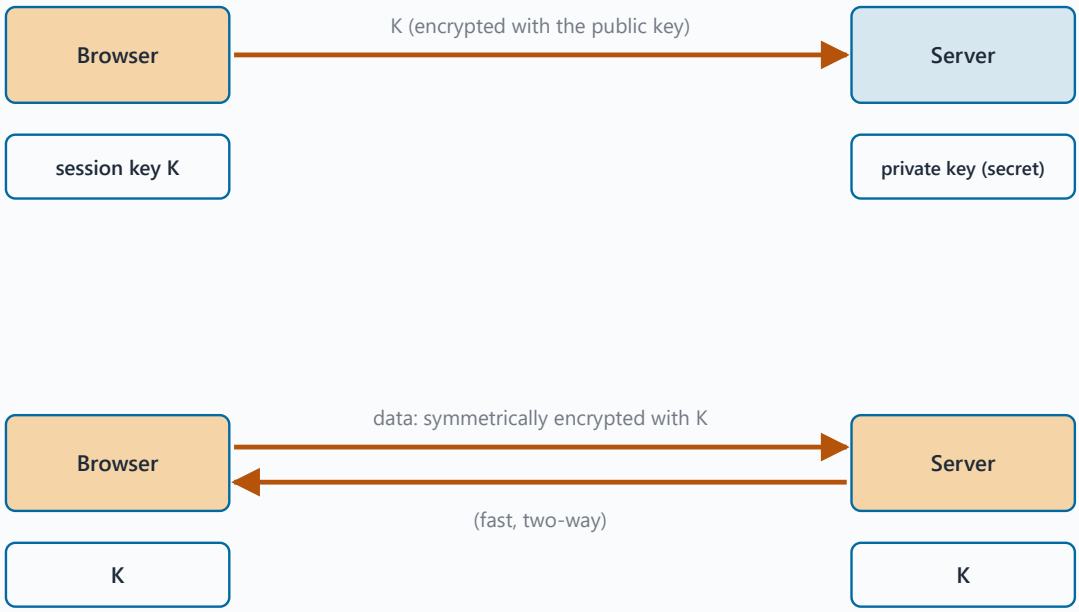


Figure 1.4.26: At the top, the asymmetric step: the browser encrypts the session key K with the server’s public key and sends it. At the bottom, the symmetric step: both sides hold K and data flows encrypted with K in both

directions.

This description is the “key transport with RSA” model used in older TLS versions, and it shows the principle simply. The current TLS 1.3 does not encrypt and send K; the two sides produce the shared key together with a Diffie–Hellman-type **key agreement**, and the key never goes onto the wire. The result is the same: the same secret key forms on both sides, and an eavesdropper cannot learn it. In this process the public key in the certificate is used not for encryption but only to prove the server’s identity: the server signs the handshake with its private key, and the browser verifies the signature with the public key in the certificate. Because asymmetric operations are hundreds of times slower than symmetric ones, this division of labor makes the cost of encryption almost unnoticeable.

1.4.5 The Web

In everyday speech, “going on the internet” often means opening a web browser. The Web, however, is only one of the services that run on the internet. This section first separates the two concepts, then follows step by step what a web page consists of and how it reaches the browser. The second half of the section is devoted to search engines and to evaluating the information found.

1.4.5.1 The internet and the Web

Definition 1.4.17

The **Web** (World Wide Web) is a service made up of pages and web applications linked to one another, accessed over HTTP/HTTPS. The internet is the infrastructure; the Web is one of the services on top of that infrastructure.

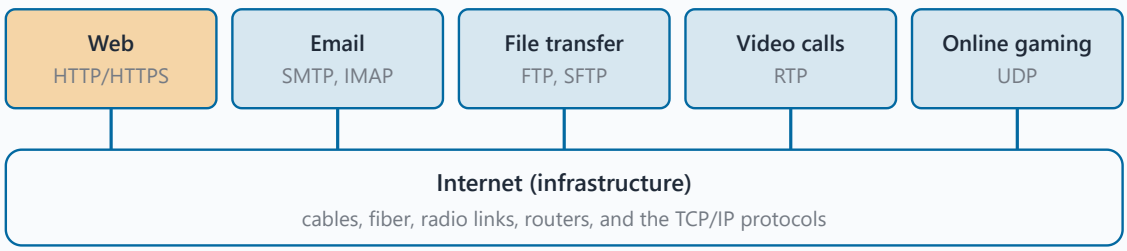


Figure 1.4.27: Services running on the internet infrastructure (cables, fiber, radio links, routers, and the TCP/IP protocols): the Web (HTTP/HTTPS), email (SMTP, IMAP), file transfer (FTP, SFTP), video calls (RTP), and online gaming (UDP).

By analogy: the internet is a country's road network, and the Web is one of the bus companies running on those roads. Freight companies, ambulances, and private cars use the same roads too. Email, file transfer, video calls, and online games do not use the Web but the internet infrastructure directly. A messaging app on a phone also runs over the internet without ever opening a browser.

> **Common mistake**
>
> Using "the internet" and "the Web" as synonyms. The internet is the network infrastructure that has been developing since 1969; the Web began in 1991 on top of that infrastructure. Email can be sent without the Web; the Web cannot run without the internet.

1.4.5.2 HTML, CSS, JavaScript, and the browser

A web page is not a single file but a collection of resources written in three languages that work together. Each language has a separate job:

- **HTML** (HyperText Markup Language) defines the page's **structure**: where the heading is, where the paragraph is, which text is a link.
- **CSS** (Cascading Style Sheets) sets the **appearance**: colors, fonts, spacing, page layout.
- **JavaScript** adds **interaction and behavior**: what happens when a button is clicked, checking a form before it is submitted, updating the page without reloading it.

The **web browser** fetches these resources from the server, interprets them, and displays them on screen. The two-line HTML source below appears in the browser as a large "Hello" heading with a paragraph beneath it (Figure 1.4.28).

```
<h1>Hello</h1>
<p>My first web page.</p>
```

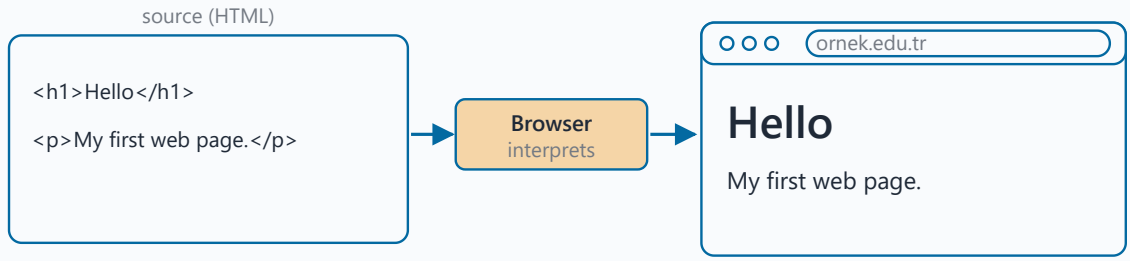


Figure 1.4.28: The HTML source on the left passes through the browser and appears as a formatted page in the window on the right.

`<h1>` and `</h1>` are **tags**: they say that the text between them is a first-level heading. `<p>` marks a paragraph. HTML is not a programming language but a **markup language**: it does no calculation and builds no loops; it only states what each part of the content is. CSS decides what size and color the heading appears in, and JavaScript decides what happens when a button is pressed.

1.4.5.3 The journey of a web request

All the concepts described so far play a part together in opening a single web page. Suppose the user types `https://www.ornek.edu.tr/announcements` in the address bar and presses Enter.

Step	What happens?	Related concept
1	The browser splits the URL into its parts: protocol https, port 443, domain name www.ornek.edu.tr, path /announcements.	URL, default port
2	The browser asks for the IP address of the domain name.	DNS resolution
3	The answer arrives: the server is at 203.0.113.10.	DNS, cache
4	The browser sets up a TCP connection with the server; because it is HTTPS, the encryption keys are agreed with TLS.	Three-way handshake, TLS
5	An HTTP request is sent over the encrypted channel: “give me the /announcements resource” (`GET /announcements`).	HTTP request
6	The server processes the request: it reads a file, or pulls the announcements from a database and generates HTML.	Server
7	The response comes back: status code 200 OK (success) and the HTML content of the page. Had the resource not been found, 404 would have been returned.	HTTP response, status code
8	While reading the HTML, the browser fetches the CSS, JavaScript, and image files it needs with separate requests.	Additional requests
9	The browser combines the HTML structure, the CSS appearance, and the JavaScript behavior and draws the page on screen.	Browser

Table 1.4.15: The journey of a web request and the concept used at each step.

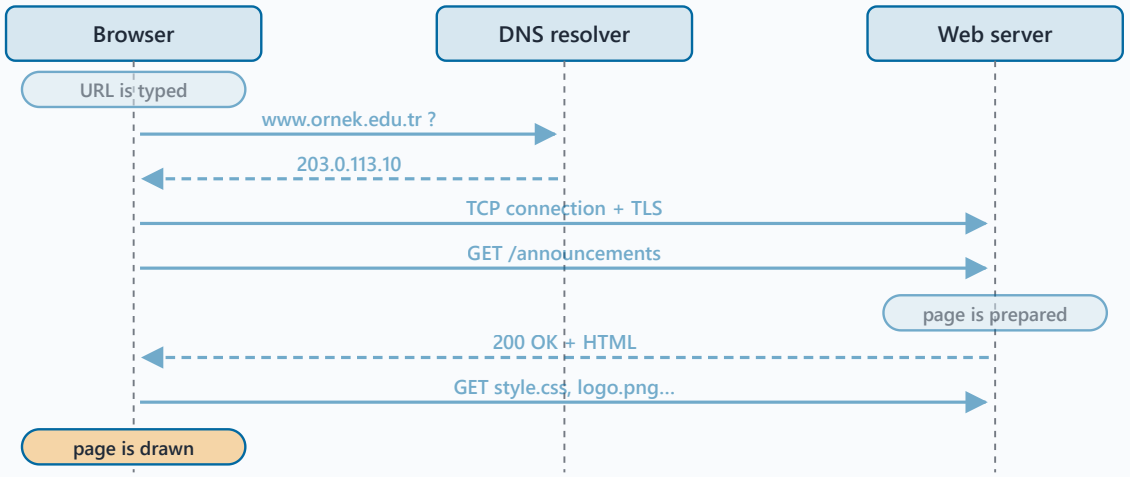


Figure 1.4.29: Timeline of a web request: the messages among the browser, the DNS resolver, and the web server in order from top to bottom; at the very bottom the page is drawn.

A single click on the surface starts at least four separate conversations in the background: a DNS query, a TCP handshake, a TLS negotiation, and an HTTP request. Each takes at least one round trip. If the server is 25 ms away (round trip), about 100 ms is spent waiting on these conversations alone before the first byte arrives. This shows why latency affects page load time so much, independently of bandwidth.

Status codes. Every response from the server carries a three-digit **status code**. The first digit tells the kind of result: codes starting with 2 mean success, with 3 redirection, with 4 a client-side error, and with 5 a server-side error.

Code	Meaning	When is it seen?
200 OK	Success	The requested resource was sent.
301	Moved permanently	The page's address has changed; the browser goes to the new address.
403	Forbidden	The resource exists, but this user has no permission to access it.
404	Not found	The path was mistyped or the page has been removed.
500	Internal server error	An error occurred while the program on the server was running.

Table 1.4.16: Commonly seen HTTP status codes.

To follow this journey step by step, you can use the The Journey of a Web Request simulator.

1.4.5.4 Search engines: crawl, index, rank

When a query is typed into a search engine, the Web is not browsed at that moment. Reading billions of pages one by one for every query would take hours. The answer comes from an **index** prepared in advance. A search engine's job can be summarized in four stages: crawl, index, rank, show results.

1. **Crawl.** A **web crawler** starts from a list of starting pages. It downloads each page, finds the links in it, and adds those links to a queue of pages to visit. The whole Web is not crawled; only pages reachable through links and that the site owner allows to be crawled are crawled.
2. **Index.** The words in the crawled pages are written into an **inverted index**.
3. **Rank.** Candidate pages that contain the words in the query are found and scored with criteria such as relevance, freshness, incoming links, and loading speed.
4. **Results.** The highest-scoring pages are listed with a title, an address, and a short snippet. All of this is usually completed in a fraction of a second.

Definition 1.4.18

An **inverted index** is a table that holds, for each word, the list of pages in which that word occurs: word → pages. It is called "inverted" because it reverses the natural arrangement of page → words. The result of a query with several words is the **intersection** of the words' page lists.

The index at the back of a book explains well how an inverted index works. To find where the word "router" occurs in a textbook, you do not have to read the book from beginning to end; the index at the back says "router: 12, 45, 78." A search engine's index does the same job at the scale of the Web.

Example 1.4.12

Suppose four pages have been crawled: Page 1 "cat food", Page 2 "cat care", Page 3 "dog food", Page 4 "cat food price". Let us build the inverted index and answer three queries: (a) `cat food` , (b) `food -cat` , (c) `dog care` .

1. The words of each page are added to the index: cat → {1, 2, 4}; food → {1, 3, 4}; care → {2}; dog → {3}; price → {4}.
2. (a) Both words must occur: {1, 2, 4} ∩ {1, 3, 4} = {1, 4}. Result: Page 1 and Page 4; ranking decides which appears first.
3. (b) "food" must occur and "cat" must not: {1, 3, 4} − {1, 2, 4} = {3}. Result: Page 3.
4. (c) {3} ∩ {2} = ∅. No page contains both words; there is no result.

During the query no page was read; only two short lists were compared. In a real index the list for the word "cat" holds millions of pages, but the intersection operation is still very fast.

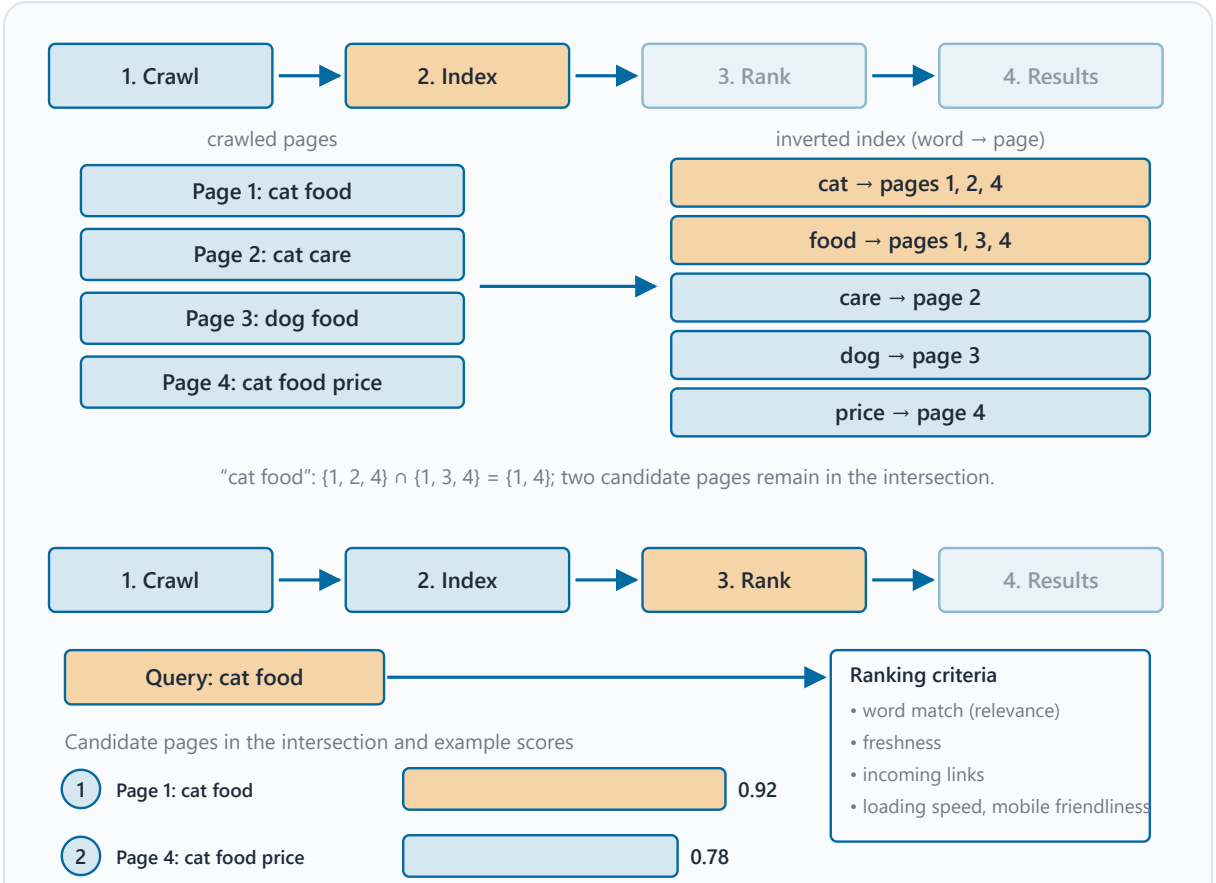


Figure 1.4.30: At the top, indexing: the words of the four crawled pages are written into the inverted index; for "cat food" the intersection is {1, 4}. At the bottom, ranking: the two candidate pages left in the intersection are scored by criteria such as relevance, freshness, and incoming links; Page 1 (0.92) comes ahead of Page 4 (0.78) (the scores are only an example).

1.4.5.5 Effective searching

Search engines recognize special **operators** for narrowing a query. Using the right operator picks out the information you want from among thousands of irrelevant results.

Method	Example	Purpose
Exact phrase	`"computational thinking"`	Search for the words side by side and in this order.
Exclusion	`java -coffee`	Remove the unwanted meaning.
Site restriction	`site:gov.tr scholarship`	Search only pages in particular domains.
File type	`filetype:pdf network security`	Restrict the document format.

Table 1.4.17: Search operators.

Operators can be combined. For example, to find PDF documents that contain the phrase “internship application” only on university sites, type `site:edu.tr filetype:pdf "internship application"`. The query `java -coffee` is also exactly what the inverted-index operations do: the pages in the “coffee” list are removed from the “java” list.

> **Common mistake**
>
> Putting operators inside quotation marks or mixing up their places. If you type `"site:gov.tr scholarship"`, this string is searched as an exact phrase and the operator does not work. A form such as `filetype:gov.tr` is also meaningless, because `filetype:` expects a file extension and `site:` expects a domain name.

1.4.5.6 Evaluating sources and AI output

When a search engine ranks a page near the top, it has not said that the page is correct. Anyone can publish on the Web; wrong, outdated, or deliberately misleading information is just as easy to find. So every piece of information found should be tested with a few questions.

Question	What to look at?
Author	Who wrote it? Is it a person with expertise on the subject, or a responsible organization?
Date	When was it published or updated? Is it current for the topic?
Source	Where does the information come from? Are citations and links given?
Verification	Can the data and claims be verified?
Cross-checking	Do other reliable sources, independent of each other, give the same information?

Table 1.4.18: Source evaluation questions.

Let us think through an example. Suppose a forum page says “Wi-Fi 6 uses the 6 GHz band.” The author is unknown, there is no date, and no source is cited. Looking at Table 1.4.4 or at the pages of the organizations that publish the standard shows that 6 GHz came with Wi-Fi 6E, not Wi-Fi 6. A single cross-check was enough to expose the error.

AI output. Chat-based AI tools produce fluent and confident text. This text can be wrong too; it can even contain facts that do not exist and sources that do not exist. The tool prioritizes whether a sentence is plausible as language, not whether it is true. So important information obtained from AI should be verified with the criteria above, just like a web page. In particular, check whether the sources given really exist and really contain that information.

1.4.6 Digital services

A large part of everyday life runs on top of the internet infrastructure and the Web: public services, communication, entertainment, shopping, file storage, and AI tools. This section describes how some of these services work: cloud service models, types of electronic commerce, and the journey of an email.

1.4.6.1 Online services

Field	Examples
Public sector	e-Devlet, health, tax, and education services.
Communication	Social networks, instant messaging, and community platforms.
Media	Video, music, and digital broadcasting services.
Commerce	Stores, marketplaces, and payment systems.
Cloud	File storage, collaboration, and applications.
Artificial intelligence	Tools that offer text, image, and decision support.

Table 1.4.19: Types of online services.

1.4.6.2 Cloud service models and CDNs

Cloud computing is renting computing power, storage, and software from a provider over the internet instead of hosting them on your own machines. There are three models, depending on how much is rented. The easiest way to tell the models apart is to list the layers an application needs, from bottom to top, and see who is responsible for each layer.

Layer	On-premises	IaaS	PaaS	SaaS
Application	User	User	User	Provider
Data	User	User	User	User (content)
Runtime	User	User	Provider	Provider
Operating system	User	User	Provider	Provider
Server and network hardware	User	Provider	Provider	Provider
Example	Own server room	Rented virtual server	Application hosting platform	Email, online office

Table 1.4.20: Division of responsibility in the cloud service models ("User": the organization or person receiving the service).

- **On-premises:** the organization sets up and runs everything itself; all responsibility, from hardware to application, rests with the organization.

- **IaaS** (Infrastructure as a Service): the provider supplies the physical machines and the network; the user installs their own operating system and software on the virtual server.
- **PaaS** (Platform as a Service): the provider also manages the operating system and the runtime environment; the user only uploads their own application.
- **SaaS** (Software as a Service): the application itself comes ready-made; the user logs in from a browser and uses it. Online office programs and web-based email are examples of this model.

Pay attention to the "Data" row of the table. Even in SaaS, the user is responsible for the *content* of the data: a document deleted by mistake, an account taken over through a weak password, or a folder needlessly shared with everyone is the user's problem, not the provider's.

CDN. A **content delivery network** (CDN) keeps copies of the same content on servers in various places around the world and serves each user from the copy closest to them. This both reduces latency and lightens the load on the origin server. Popular video sites and large news sites distribute their content this way.

Example 1.4.13

A user in Giresun connects to a site whose origin server is in a city about 2500 km away. The site also uses a CDN server in Istanbul (about 800 km away). Taking the speed of light in fiber as 200,000 km/s, let us compare only the path latency.

1. Origin server: one way 2500 ÷ 200,000 = 0.0125 s = 12.5 ms; round trip 25 ms.
2. CDN server: one way 800 ÷ 200,000 = 0.004 s = 4 ms; round trip 8 ms.
3. If at least four round trips are needed to open a page (DNS, TCP, TLS, HTTP): at least 4 × 25 = 100 ms on the origin server, at least 4 × 8 = 32 ms on the CDN.

Because cables never follow a straight line and there is queuing in routers, real times are longer; but the ratio is roughly preserved. Shortening the distance gives a gain that widening the line cannot.

1.4.6.3 E-commerce

Electronic commerce (e-commerce) is buying and selling products and services over the internet. Four types are distinguished by who the parties are. In the abbreviations, B stands for business and C for consumer; the first letter indicates the seller and the last the buyer.

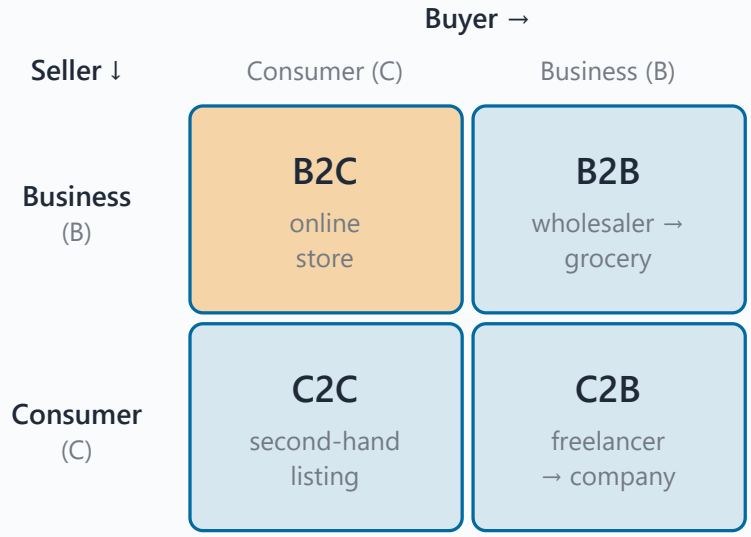


Figure 1.4.31: The four types of e-commerce: the seller (rows) and the buyer (columns) can each be a business or a consumer.

- **B2C** (business to consumer): buying shoes from a store's website.
- **B2B** (business to business): a restaurant placing an online order with a wholesaler.
- **C2C** (consumer to consumer): a student selling an old book to another student on a second-hand listing site.
- **C2B** (consumer to business): a freelance designer designing a logo for a company, or a photographer selling photos to an agency.

Whatever the type, a few things should be checked before the transaction: the seller's identity and contact information, the final price including shipping and taxes, the delivery time, the return terms, and the security of the payment page. The payment page should appear with the correct domain name in the address bar and with HTTPS. "Unmissable deals" far below the market price are among the most common signs of fraud.

1.4.6.4 The email address and its parts

An email address consists of two parts and a sign that separates them. In the address `sercan.kulcu@giresun.edu.tr`:

- The **username** (`sercan.kulcu`) identifies the mailbox.
- The **@ sign** separates the username from the domain name.
- The **domain name** (`giresun.edu.tr`) indicates the organization that runs the mail service. To find out which server to deliver the message to, the sending server asks DNS about this part.

The fields filled in while writing a message determine who the message reaches and how.

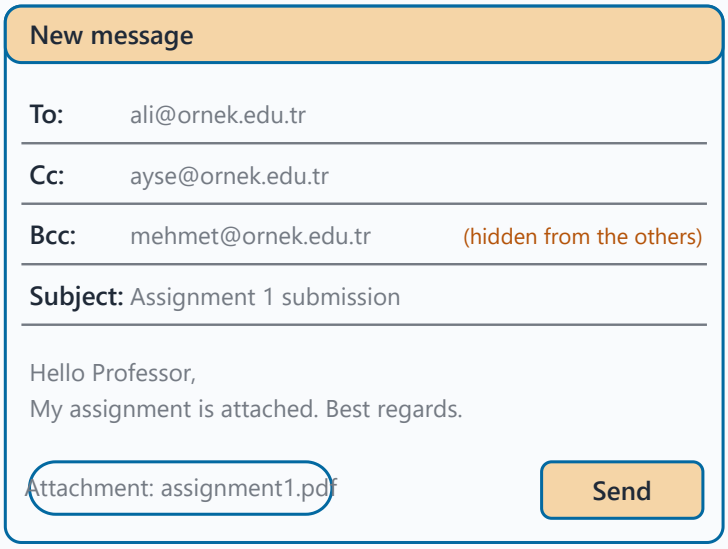


Figure 1.4.32: The new message window: To, Cc, and Bcc recipients, subject, message text, attachment, and Send button. A Bcc recipient is not visible to the other recipients.

Field	Meaning
To	The main recipient of the message.
Cc	People who receive the message for information; all recipients can see them.
Bcc	Recipients that the other recipients cannot see.
Subject	A short, descriptive title for the message.
Attachment	A file sent with the message.
Reply / Forward	Replying to the sender or passing the message on to someone else. “Reply all” goes to all the visible recipients.

Table 1.4.21: Email components.

Example 1.4.14

Elif, the department secretary, sends an announcement like this: To: Can · Cc: Deniz and Ece · Bcc: the department chair. If Can chooses “Reply all,” who does the reply go to? Whom does Ece see in the recipient list?

1. Each recipient sees the To and Cc fields in the message header; the Bcc field stays only with the sender.
2. The list Ece sees: To: Can; Cc: Deniz and Ece. The department chair is not in the list.
3. If Can chooses “Reply all,” the reply goes to the sender and all the visible recipients: Elif, Deniz, and Ece. Because the department chair does not appear in the list, they do not receive the reply.

The Bcc field is also used to hide recipients' addresses from one another when sending an announcement to large groups. An announcement sent to a hundred people in the Cc field reveals all hundred addresses to everyone.

1.4.6.5 How does an email arrive?

Email is a separate service that works with protocols different from the Web's. Sending is done with **SMTP** and reading with **IMAP**. In between there are at least two mail servers and a DNS query. In the example, Ali sends a message to the address `ayse@ornek.com.tr`.

1. Ali writes the message and presses Send.
2. Ali's email client hands the message over SMTP to his organization's outgoing mail server (`smtp.ornek.edu.tr`).
3. The server asks DNS for the mail server of the domain name after the @ sign (`ornek.com.tr`). This question is an **MX record** (mail exchanger) query.
4. DNS answers: mail arriving for ornek.com.tr is accepted by `mx.ornek.com.tr`.
5. The message is transferred over SMTP to the recipient's mail server.
6. The receiving server passes the message through the spam and virus filter and stores it in Ayşe's **mailbox**.
7. Ayşe reads the message over IMAP. Because the message stays on the server, it looks the same on her phone and her computer.

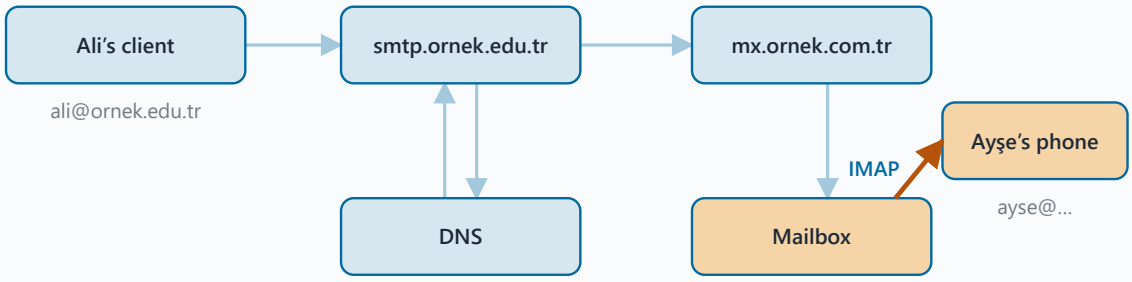


Figure 1.4.33: The path of an email: Ali's client → smtp.ornek.edu.tr (asks DNS for the MX record) → mx.ornek.com.tr → filter → Ayşe's mailbox → Ayşe's phone over IMAP.

Common mistake

Thinking that SMTP is also used to read email. SMTP is only for sending and for transferring between servers. IMAP is used to read the messages in a mailbox. (The older POP3 protocol is also for reading, but it mostly downloads messages from the server and deletes them; so it is

not suitable for use with more than one device.)

1.4.7 Security

The openness of the internet is the feature that makes it so useful; the same openness also opens the door to malicious people. Most attacks exploit people's carelessness and habits more than technical flaws. This section first defines what security tries to protect, then takes up the threats and the measures that can be taken against them in turn.

1.4.7.1 The three goals of information security

Definition 1.4.19

Information security tries to protect three goals; from their initials, this triad is called the **CIA** triad.

- **Confidentiality:** Only authorized people can access the information.
- **Integrity:** The information is not changed without permission.
- **Availability:** The information and service are reachable when needed.

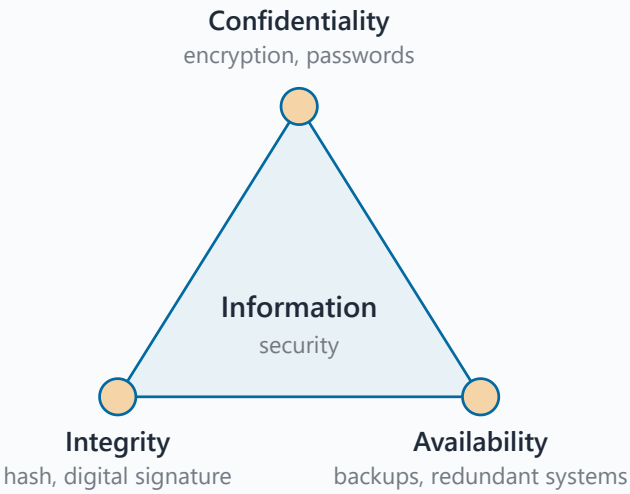


Figure 1.4.34: The three goals of information security and the typical protection method for each: encryption and passwords for confidentiality, hashes and digital signatures for integrity, backups and redundant systems for availability.

Each goal has its own threats. A student's password being stolen and their transcript read by someone else is a **confidentiality** breach. An attacker breaking into the student information system and changing a grade is an **integrity** breach; even if the information stayed secret, it is no longer correct. Ransomware locking the files on a server, or a system being brought down by overload, is an **availability** breach; nobody has stolen or changed the information, but it cannot be reached.

1.4.7.2 Personal data, KVKK, and e-Devlet

Definition 1.4.20

Personal data is any information relating to an identified or identifiable person: name, ID number, phone, address, photo, location, health information. A **digital footprint** is the sum of the records left by online activity. **Privacy** is being able to control with whom, for what purpose, and for how long information is shared.

There are two kinds of digital footprint (Figure 1.4.35). An **active footprint** is content shared on purpose: posts, photos, comments, forms filled in. A **passive footprint** is left without noticing: location history, cookies, IP address, search history. The second kind is often more extensive, and the user has less control over it.

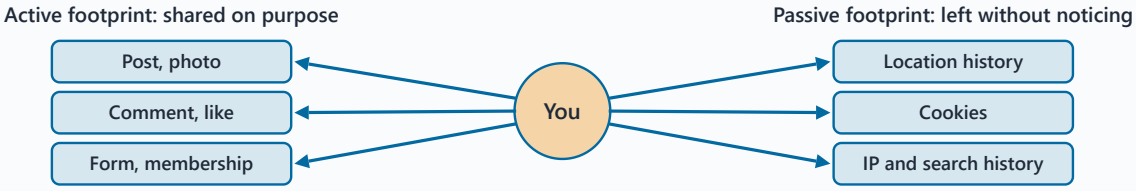


Figure 1.4.35: The digital footprint. On the left, the active footprint (what is shared on purpose: post, photo, comment, like, form, membership); on the right, the passive footprint (what is left without noticing: location history, cookies, IP and search history).

Information that has gone onto the internet once can often not be taken back entirely: it may have been copied, screenshotted, or archived. So three questions should be asked before sharing information: Is it necessary to share this? Who will be able to see it? Can it be deleted later? For example, sharing a vacation photo "right now" announces to everyone that the house is empty.

KVKK. In Turkey the protection of personal data is regulated by **Law No. 6698 on the Protection of Personal Data** (KVKK), in force since 2016. The law protects people's rights in the processing of personal data. The organization or company that processes the data, the **data controller**, is obliged to explain for what purpose the data is processed and to keep the data secure. A person in turn has the right to learn which data about them is being processed and to ask for wrong data to be corrected.

e-Devlet security. Public services are provided through e-Devlet with identity verification. Because an e-Devlet account opens the door to land registry, tax, health, and education records, it is among the accounts that most need protecting. A strong password used nowhere else, two-step verification, and not sharing the password with anyone are the basic rules. Do not click links in emails or SMS messages that ask for identity information; reach the official site by typing the address into the browser by hand.

1.4.7.3 Malware

Malware is the general name for programs written to harm a computer or its user. Types are distinguished by how they spread and what they do.

Type	Basic behavior	Typical example
Virus	Infects files; spreads when the user opens an infected file.	Running an infected program from a USB stick.
Worm	Can spread by itself over the network; needs no user action.	A program that exploits a flaw in a service that has not been updated and moves to other machines on the network.
Trojan horse	Looks useful but hides a harmful function.	A remote access program hidden inside a pirated game offered “for free.”
Ransomware	Encrypts files to make them inaccessible and demands payment to open them.	All documents becoming unopenable and a payment demand appearing on screen.
Spyware	Secretly watches the user’s activities and sends them out.	A program that records keystrokes and sends the passwords to the attacker.

Table 1.4.22: Types of malware.



Figure 1.4.36: The difference in how a virus and a worm spread. On the left, the virus: the infected file passes to other files when the user opens or shares it. On the right, the worm: it spreads by itself from PC 1 over the network to PC 2 and PC 3.

The difference between a virus and a worm is how they spread: a virus waits for a user to open a file; a worm does not wait at all and finds exposed machines on the network itself. That is why worms can spread to thousands of machines in a very short time. The most effective protection is to keep the operating system and applications up to date; most of the flaws worms use were closed months earlier by updates.

1.4.7.4 Social engineering and phishing

Definition 1.4.21

Social engineering is a method of obtaining information or getting an action carried out by using people's trust, fear, or curiosity instead of technical flaws. **Phishing** is an attack that uses fake messages posing as a trusted organization to capture a user's password, card details, or verification code.

A phishing attack proceeds like a chain with four links: an urgent or tempting message arrives, the message contains a fake link, the user enters their information on the page behind the link, and the account is taken over. At each link the attacker has a move and the user has a point where the chain can be broken.

Link	What does the attacker do?	How is the chain broken?
1. Fake email	Sends an official-looking, urgent, and threatening message; the sender's domain is not the organization's.	Check the sender address and domain; be suspicious, do not click, report it.
2. Fake page	The link leads to a login page that closely resembles the real one; the domain is different.	Type the address by hand; the password manager will not fill in the password on the wrong domain.
3. Password entry	What is typed goes straight to the attacker's server; the victim is often redirected to the real site and does not notice.	Before entering a password, verify with the organization through a channel you already know.
4. Account takeover	Logs in to the account, changes the password, tries the same password on other accounts, and sends new phishing messages to the contacts.	Multi-factor authentication is on; even if the password is stolen, the login is refused. Every account has a different password.

Table 1.4.23: The phishing attack chain and the breaking point at each link.

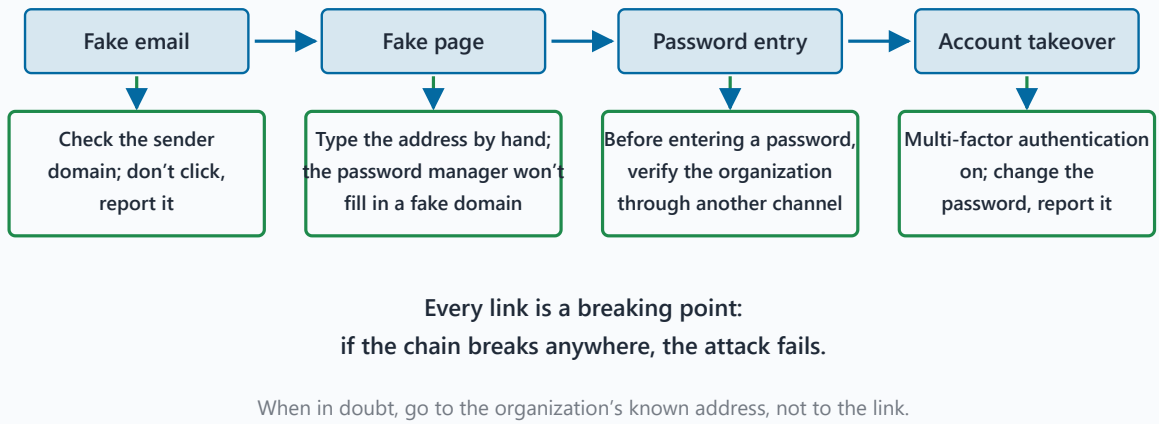


Figure 1.4.37: The four links of the phishing chain and the breaking point under each link. If a single link is broken, the attack fails.

The chain analogy has an important consequence: to be protected, you do not need to be perfect at all four links; breaking a single link is enough. A careful user recognizes the message at the first link. If they miss it, the habit of typing the address by hand protects them at the second link. If they enter the password anyway, multi-factor authentication saves the account at the fourth link.

Phishing is not done only by email. Phishing by SMS ("Your parcel could not be delivered, update your address details"), phishing by voice call ("We are calling from your bank; could you read out the code that came to your phone?"), and phishing through social media messages are also common. The common signs are the same: urgency, fear, or a tempting offer, and getting you to click a link or hand over a code.

1.4.7.5 Examining a phishing email

The sample message appears to have been sent on behalf of a university's IT department. There are six separate warning signs in the message (Figure 1.4.38).

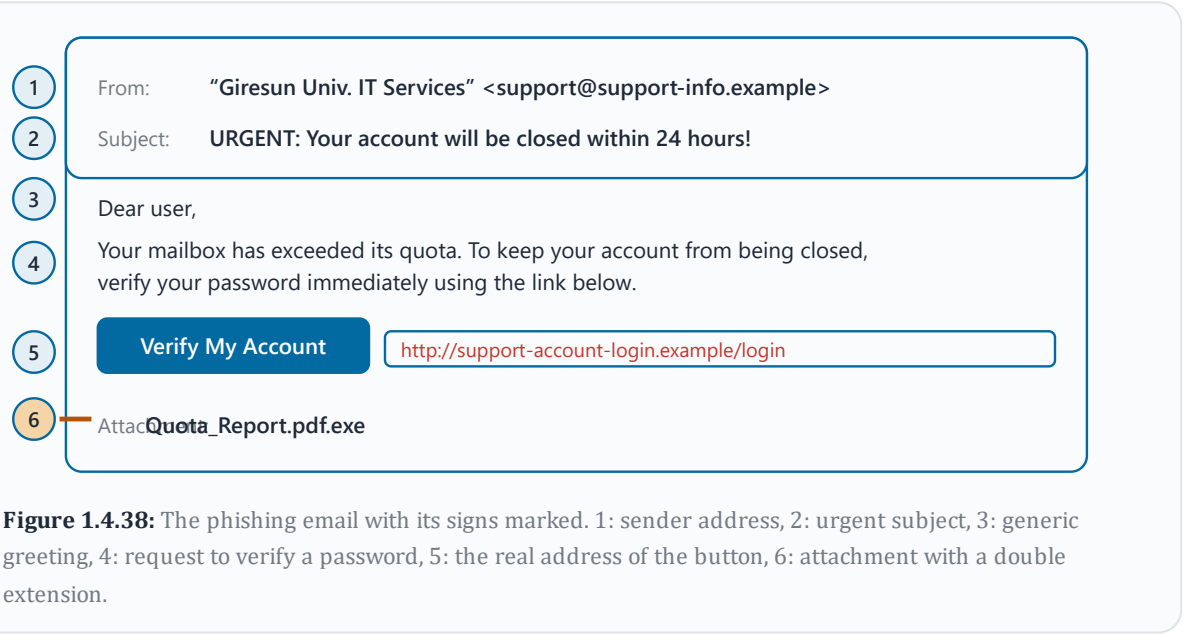


Figure 1.4.38: The phishing email with its signs marked. 1: sender address, 2: urgent subject, 3: generic greeting, 4: request to verify a password, 5: the real address of the button, 6: attachment with a double extension.

1. **Sender address.** The display name is "Giresun Univ. IT Services," but the address is `support@support-info.example`. The domain of a genuine message should have been `giresun.edu.tr`. The display name can be written as anyone likes; the part to look at is the part of the address after the @.
2. **Urgency and threat.** The subject line says "URGENT: Your account will be closed within 24 hours!" The aim is to make the user click without thinking.
3. **Generic greeting.** It says "Dear user." An organization knows its own student or employee by name.
4. **Request to verify a password.** Real organizations do not "verify" a password by email or through a link.

5. **The real address of the link.** The address that appears when you hover over the button (without clicking) goes to the domain `support-account-login.example`; it has nothing to do with the organization.
6. **Attachment with a double extension.** The attachment's name is `Quota_Report.pdf.exe`. Because the last extension is `.exe`, this is not a document but an executable program. If Windows is set to hide extensions, the user sees only "Quota_Report.pdf."

Even one of these signs is enough to make a message suspicious. When you meet such a message, the safest course is to use none of the links in it and instead go to the organization's official site by typing the address by hand, or to call the organization on a number you already know. If the announcement is real, it will also appear on the official site.

Common mistake

Considering it enough that the organization's name appears in the address of the link. The address `giresun.edu.tr.account-verify.example` belongs not to Giresun University but to the `account-verify.example` domain. A domain name is read from right to left (1.4.3.6); what decides is the registered domain at the far right.

1.4.7.6 Password strength

An attacker can try all possible passwords in turn to guess a password; this is called a **brute-force** attack. How long the attack takes depends on the number of possible passwords. If the number of symbols that can be used in the password is S and the password's length is L, the number of possible passwords is

$$N = S^L$$

There are S options for each character, and the choices for the L characters are independent of one another; that is why the options are multiplied. If the attacker can make D guesses per second, the worst-case time is $N \div D$ seconds. In these notes $D = 10^{10}$ is used, that is, 10 billion guesses per second; this is a realistic value for an attacker who tries hashed passwords with powerful graphics cards.

Example 1.4.15

For an attacker making 10^{10} guesses per second, let us compute the worst-case time for three kinds of password: (a) 8 lowercase letters, (b) 8 characters containing letters, digits, and symbols, (c) a passphrase made of 16 lowercase letters. The English alphabet has 26 lowercase letters; all the printable characters on the keyboard number 94.

1. (a) $N = 26^8 \approx 2.1 \times 10^{11}$. Time: $2.1 \times 10^{11} \div 10^{10} \approx 21$ seconds.

2. (b) N = $94^8 \approx 6.1 \times 10^{15}$. Time: 6.1×10^5 s. Since a day has 86,400 s, $6.1 \times 10^5 \div 86{,}400 \approx 7$ days.
3. (c) N = $26^{16} \approx 4.4 \times 10^{22}$. Time: 4.4×10^{12} s. Since a year has about 3.16×10^7 s, that is $\approx 1.4 \times 10^5$ years, or about 140,000 years.

Raising the variety of symbols from 26 to 94 raised the time from 21 seconds to 7 days (about 29,000 times). Raising the length from 8 to 16 took lowercase letters alone to 140,000 years. Length is far more effective than adding symbols.

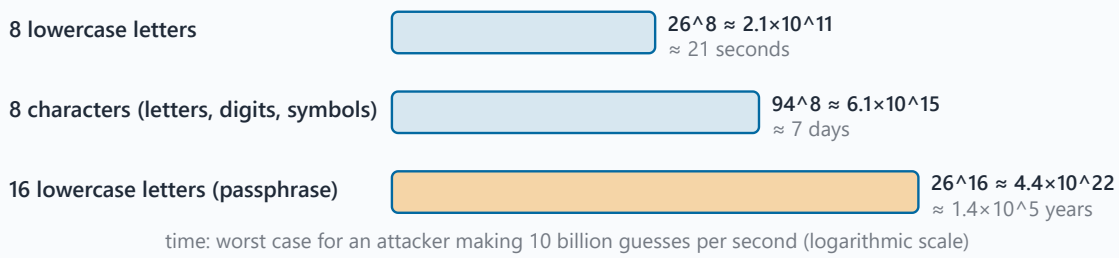


Figure 1.4.39: The number of possible passwords for three kinds of password and the worst-case time for an attacker making 10 billion guesses per second. The bars are on a logarithmic scale: the length of each bar is proportional to the number of digits in the number of possible passwords.

The formula shows why length is so effective. Increasing S enlarges the base; increasing L enlarges the exponent. Each character added to the exponent multiplies the number of possibilities by S. Adding one more character to 8 lowercase letters multiplies the number by 26; adding four more characters multiplies it by $26^4 \approx 457{,}000$. A 12-letter lowercase password ($26^{12} \approx 9.5 \times 10^{16}$) offers about 16 times as many options as an 8-character mixed password (94^8), and it is also easier to remember.

The calculation rests on an assumption: the attacker tries all the possibilities in a random order. Real attackers first try dictionary words, commonly used passwords (`123456`, `qwerty`), birth dates, team names, and small variations of these (`Galatasaray1905!`). Passwords like these are found within seconds even if they look long and mixed. A password that was used on another site and leaked is no longer secure at all: the attacker puts it among the first passwords they try.

A **password manager** is the practical solution to these problems. It generates a long, random password for each account, stores it in an encrypted vault, and fills it in when needed. The user remembers only the single master password of the vault. A password manager does not fill in the password on a phishing page, because the saved domain name does not match the page's domain name; this gives extra protection.

1.4.7.7 Multi-factor authentication

Definition 1.4.22

Multi-factor authentication (MFA) is requiring at least two different kinds of proof for login. The kinds of proof are: something the user **knows** (password, PIN), something the user **has** (phone, hardware key), and something the user **is** (fingerprint, face).

1. The user enters their username and password. This is the first factor: something they know.
2. If the password is correct, the server asks for a second proof. The authenticator app on the phone generates a 6-digit code that changes every 30 seconds. This is the second factor: something they have.
3. The user enters the code; the server verifies the code and the session opens.
4. Even if the password has leaked, the login is mostly refused because the attacker does not have the user's phone.

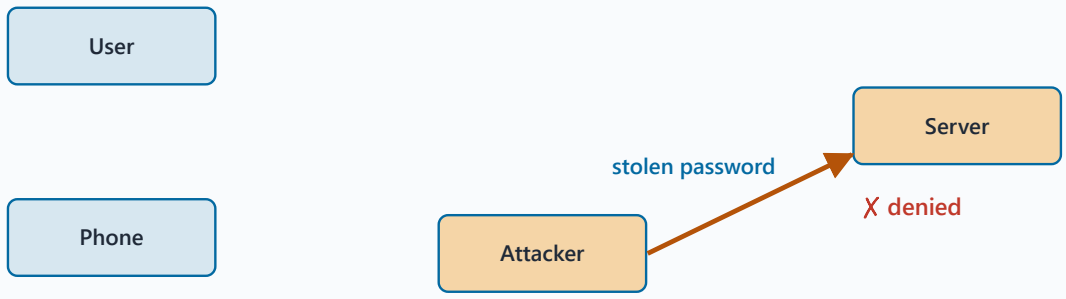


Figure 1.4.40: A login attempt with a stolen password. The attacker knows the password, but cannot produce the code on the phone, so the server refuses the login.

MFA is not perfect. In **real-time phishing**, the attacker enters the code the user typed into the fake page on the real site at the same moment. In a **SIM swap** attack, the attacker tricks the operator into moving the victim's phone number to their own SIM card and receives the codes sent by SMS. That is why an SMS code is a weaker second factor than an authenticator app. The most resistant methods are **FIDO2/passkeys** and **hardware keys**: these methods check the site's domain name themselves and hand nothing over to a fake page.

1.4.7.8 VPNs

Definition 1.4.23

A **VPN** (virtual private network) is a service that sets up an encrypted **tunnel** over the internet between a device and a VPN server. All of the device's traffic goes through this tunnel to the VPN server; the server forwards the traffic from there to its real destination.

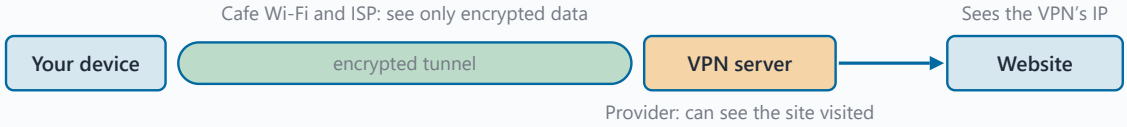


Figure 1.4.41: The VPN tunnel. The cafe Wi-Fi and the ISP see only encrypted data; the website sees the IP address of the VPN server, not the user's; the VPN provider, however, can see which sites the user visits.

It is important to separate what a VPN does from what it does not do:

A VPN does	A VPN does not
Makes local eavesdropping harder on public Wi-Fi.	Provide absolute anonymity; the VPN provider can see which sites the traffic goes to.
Shows sites the VPN server's IP address instead of the user's IP address.	Stop sites from recognizing the user if they are logged in to an account or the cookies are still there.
Gives secure remote access to an organization's network (for example, to library databases from off campus).	Block malware or phishing.

Table 1.4.24: What a VPN does and does not do.

Using a VPN means moving the traffic from being seen by the ISP or the cafe network to being seen by the VPN provider. Providers' "we keep no logs" claims are their own statements and are hard to verify from outside. So a VPN requires choosing a trustworthy provider. A VPN does not replace HTTPS either: the tunnel runs only as far as the VPN server; the part from the server to the site is encrypted only if HTTPS is used. The two complement each other.

1.4.7.9 Firewalls

Definition 1.4.24

A **firewall** is software or hardware that allows or blocks incoming and outgoing connections according to rules. The rules mostly rely on the IP address and port number and are tried **from top to bottom in order**: the first matching rule is applied, and the later ones are not looked at.

Let the rules of a firewall protecting a web server be as in Table 1.4.25. The administrator's IP address is 192.0.2.50.

No.	Direction	Destination port	Source	Action
1	Inbound	443 (HTTPS)	Everyone	Allow (web server)
2	Inbound	22 (SSH)	Admin IP only	Allow
3	Inbound	23 (Telnet)	Everyone	Block (unencrypted, legacy protocol)
4	Inbound	All others	Everyone	Block (default deny)

Table 1.4.25: A firewall rule example.

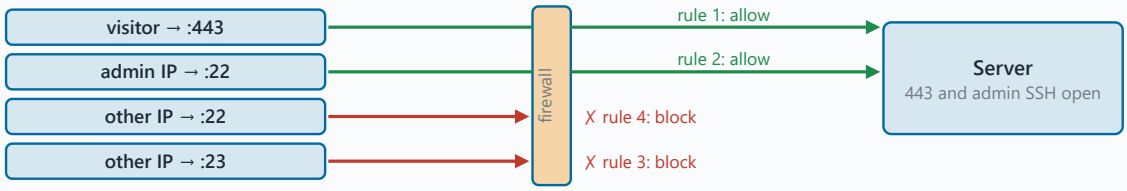


Figure 1.4.42: Four incoming connections and the firewall's decisions: the visitor's connection to 443 and the administrator's connection to 22 pass; another IP's connection to 22 is blocked by rule 4, and its connection to 23 by rule 3.

Example 1.4.16

For the following incoming connections, which rule matches and what is the decision? (a) 198.51.100.7 → 443, (b) 192.0.2.50 → 22, (c) 198.51.100.7 → 22, (d) 192.0.2.50 → 3389 (remote desktop).

1. (a) Rule 1's port (443) matches and the source is "everyone": **rule 1, allow**.
2. (b) Rule 1's port does not match. Rule 2's port (22) and source (the admin IP) match: **rule 2, allow**.
3. (c) Rule 2's port matches but the source is not the administrator; the rule does not apply. Rule 3's port is 23, so it does not match. Rule 4 covers everything: **rule 4, block**.
4. (d) Even if the source is the administrator, there is no rule that allows 3389. The ports of rules 1, 2, and 3 do not match: **rule 4, block**.

Case (d) shows the "default deny" principle: everything not explicitly allowed is blocked. Even the administrator can connect only to a port opened for them.

This is good practice: block everything first, and allow only the necessary ports. Telnet (23) carries the password in plain text because it works unencrypted; that is why it is explicitly closed. SSH (22) is needed for administration but is opened only to the administrator's address. The order of the rules matters too: if rule 4 (block everything) were written at the top of the list, no connection could ever reach rules 1 and 2.

A firewall has a limit: it restricts entry to the network, but it cannot by itself stop a harmful file that the user downloads and runs of their own accord. If the user downloads a Trojan horse from a website (over 443, that is, over an allowed connection), the firewall sees it as ordinary web traffic.

1.4.7.10 Backups: the 3-2-1 rule

Disk failure, theft, fire, accidental deletion, or ransomware… There are many ways to lose data, and against some of them no measure gives full protection. The last line of defense is the backup.

Definition 1.4.25

The **3-2-1 backup rule**: keep at least **3 copies** of the data (the original and at least two backups); keep the copies on at least **2 different types of media**; and keep at least **1 of the copies in a different location** (the cloud or another building).

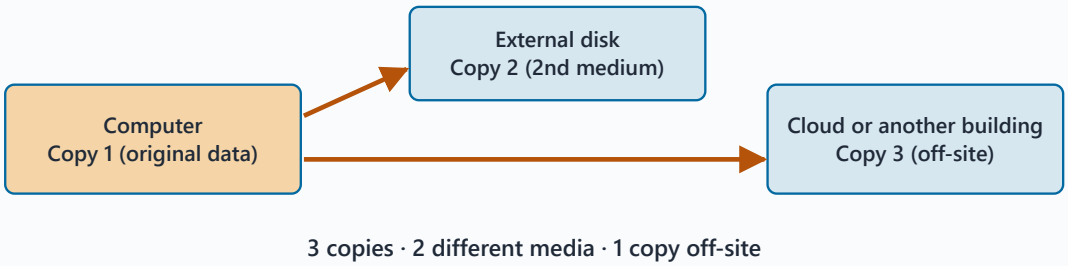


Figure 1.4.43: The 3-2-1 rule: the original data on the computer (copy 1), the backup on an external disk (copy 2, the second medium), and the backup kept in the cloud or in another building (copy 3, off-site).

Each part of the rule guards against a different threat. **Three copies** guard against the chance that one copy is corrupted. **Two media** guard against two media of the same kind (two disks from the same batch, for example) failing at the same time for the same reason. **One copy off-site** guards against events that affect the whole room, such as fire, theft, or flooding.

Example 1.4.17

Mert keeps his project files on his laptop. Every evening he copies the files to a USB stick in his desk drawer; he also makes a copy in a second folder on the laptop. Let us assess the situation against the 3-2-1 rule.

1. Number of copies: the original files on the laptop, the second folder on the laptop, the USB stick. There are three copies.
2. Number of media: the laptop disk and the USB stick. There are two different media. But because the two copies on the laptop sit on the same disk, both go together when the disk fails; that copy does not count as separate in terms of media.

3. Off-site copy: there is none. If the bag is stolen or the room burns, the laptop and the USB stick are lost together.

What is missing is a copy in a different location. Backing the files up to a cloud storage service or to a disk at home completes the rule.

Two more warnings are in order. First: a backup is useful only if it can be restored. It has often happened that backups taken for years could not be opened on the day they were needed; so you should occasionally restore a file from the backup to test it. Second: a disk left permanently connected to the computer can be encrypted along with the original files when ransomware strikes. Connecting the backup disk only while the backup is being made reduces this risk. Cloud folders that synchronize carry a similar risk: a file deleted or encrypted by mistake can be mirrored to the copy in the cloud as well. A service that keeps version history lets you go back to an older version in this case.

1.4.8 Summary

- The **internet** is the decentralized global network of networks that joins networks with common protocols (TCP/IP). Turkey connected to the internet on 12 April 1993 with the METU–NSFNET line.
- A user connects to the internet through an **ISP** and an access line (fiber, DSL, mobile, satellite). **Bandwidth** is the bits carried per second, **latency** is the arrival time of the first bit; download time = MB × 8 ÷ Mbit/s.
- A **client** asks, a **server** answers, and a **protocol** sets the rules. A **switch** works by MAC address inside a local network, and a **router** works by IP address between networks. By scope, networks are PAN, LAN, MAN, and WAN; in Wi-Fi, speed rises and range falls as frequency rises; the current security protocols are WPA2 and WPA3.
- Protocols work in four layers, and each layer adds its own header (**encapsulation**). Data is split into **packets**; packets can arrive by different paths and out of order, and ordering and resending are TCP's job.
- A router chooses the **longest prefix** among the rows that cover the destination and decrements the **TTL** by 1; when the TTL reaches zero, the packet is dropped.
- An **IPv4** address is 32 bits (four octets, 0–255); an **IPv6** address is 128 bits. Private addresses (10/8, 172.16/12, 192.168/16) are not routed on the internet; the router translates them with **NAT** to its own public address and a port.
- **DNS** translates a domain name to an IP address: the resolver descends from the root to the top-level domain and the authoritative server, and keeps what it learns in its cache for the TTL. A domain name is read from right to left.
- A **URL** consists of protocol, domain name, port, path, query, and fragment; the fragment is not sent to the server. A **port** selects the service: 80 HTTP, 443 HTTPS, 22 SSH, 53 DNS.

- **TCP** is set up with a three-way handshake and closed with four segments; ack = last byte received + 1. **HTTPS** encrypts with TLS and verifies the server's identity with a certificate; the lock does not guarantee the site's honesty.
- **Symmetric encryption** is fast, and **asymmetric encryption** solves key distribution; HTTPS uses the two together. A **hash** is not encryption and cannot be reversed.
- The **Web** is one of the services on the internet; a page consists of HTML, CSS, and JavaScript. A **search engine** crawls in advance, builds an **inverted index**, and intersects the lists at query time. Information found and AI output are tested by author, date, source, and cross-checking.
- The cloud models are IaaS, PaaS, and SaaS; a CDN reduces latency. Email is sent with SMTP, the recipient's server is found through the MX record, and the mail is read with IMAP.
- Information security protects **confidentiality, integrity, and availability**. **Phishing** is a chain of four links, and breaking a single link stops the attack. The number of possible passwords is S^L; length matters more than symbols. A password manager, **multi-factor authentication**, updates, default deny in the firewall, and **3-2-1 backups** are the basic measures; a VPN tunnel encrypts but does not provide anonymity.

1.4.9 Key terms

computer network Internet Internet service provider (ISP) bandwidth latency client server protocol switch router modem firewall local area network (LAN) wide area network (WAN) Wi-Fi layer header packet segment sequence number MAC address routing table longest prefix match default gateway time to live (TTL) IP address octet private IP address public IP address network address translation (NAT) domain name DNS resolver root server authoritative server cache top-level domain Uniform Resource Locator (URL) port three-way handshake acknowledgment (ACK) Transport Layer Security (TLS) certificate symmetric encryption asymmetric encryption hash session key World Wide Web web browser status code web crawler inverted index cloud computing content delivery network (CDN) e-commerce mail server information security personal data digital footprint malware ransomware social engineering phishing password manager multi-factor authentication passkey virtual private network (VPN) backup

---

1.4.10 Test Yourself

1. How many minutes does a 750 MB video take to download in the best case on a 25 Mbit/s line?

What result do you get if you forget to multiply by eight?

Answer

750 MB = 750 × 8 = 6000 Mbit; 6000 ÷ 25 = 240 s = 4 minutes. If you forget to multiply by eight, you get 750 ÷ 25 = 30 s; the result comes out 8 times too short.

2. What is the binary form of the address 10.20.30.40? Is this a private (local) address?

Answer

10 = 8 + 2 → 00001010; 20 = 16 + 4 → 00010100; 30 = 16 + 8 + 4 + 2 → 00011110; 40 = 32 + 8 → 00101000. The address: `00001010.00010100.00011110.00101000`. Because the first octet is 10, it is in the 10.0.0.0/8 block; it is a private address and is not routed on the internet.

3. A router's table: 10.0.0.0/8 → A, 10.1.0.0/16 → B, 10.1.2.0/24 → C, 0.0.0.0/0 → D. To which neighbor do packets destined for 10.1.2.7, 10.1.9.9, 10.200.0.1, and 8.8.8.8 go?

Answer

10.1.2.7 matches all four rows; the longest prefix is /24 → **C**. 10.1.9.9 matches /8, /16, and /0 (the third octet 9 does not match /24) → **B**. 10.200.0.1 matches only /8 and /0 → **A**. 8.8.8.8 matches only the default route → **D**.

4. A packet is sent with TTL = 5; the destination is 7 routers away. What happens?

Answer

Each router decrements the TTL by 1: it becomes 4 at the 1st router, 3 at the 2nd, 2 at the 3rd, 1 at the 4th, and 0 at the 5th. The 5th router drops the packet and returns a "time exceeded" message to the sender. The packet does not reach the destination. traceroute discovers the path by using these messages.

5. In the address `https://course.ornek.edu.tr:8443/assignment/list?week=5&class=b#submit`, which port is connected to? Which part is sent to the server?

Answer

Because the port is written explicitly, **8443** is used instead of HTTPS's default 443. The domain name is course.ornek.edu.tr, and it is translated to an IP by DNS. The request sent to the server consists of the path and the query: `/assignment/list?week=5&class=b` (two name=value pairs). The `#submit` fragment is not sent to the server.

6. A resolver's cache has the records "the .tr server" and "the authoritative server of ornek.edu.tr," and no others. How many external queries are needed for the questions `mail.ornek.edu.tr` and `www.other.edu.tr`?

   Answer

   `mail.ornek.edu.tr`: because the authoritative server of ornek.edu.tr is known, it is asked directly; **1 query**. `www.other.edu.tr`: the authoritative server of other.edu.tr is not known, but the .tr server is; the root is skipped, and first .tr and then the authoritative server are asked; **2 queries**.

7. The client's initial sequence number is 7000. After the three-way handshake it sends 500 bytes of data and closes the connection. What are the ack value in the server's SYN + ACK, the seq value of the data segment, the acknowledgment of the data, and the acknowledgment given to the client's FIN?

   Answer

   SYN + ACK: ack = 7001 (SYN uses up one number). Data: seq = 7001; bytes 7001–7500. Acknowledgment: ack = 7501. The client's FIN has seq = 7501; its acknowledgment is ack = 7502, because FIN also uses up one number.

8. For an attacker making 10^{10} guesses per second, what are the worst-case times for a 10-character password made only of digits and for a password of 10 lowercase letters?

   Answer

   Digits: 10^{10} possibilities; $10^{10} \div 10^{10} =$ **1 second**. Lowercase letters: $26^{10} \approx 1.41 \times 10^{14}$ possibilities; $\approx 1.41 \times 10^{4}$ s $\approx$ **3.9 hours**. When the variety of symbols rises from 10 to 26 at the same length, the time rises about 14,000 times; but neither counts as strong. For a strong password, the length must be increased.

9. In each of the following events, which goal of information security has been violated? (a) A hospital's appointment system is overloaded with fake requests and does not open for hours. (b) An unencrypted USB stick containing patient records is left on a bus. (c) An attacker changes product prices on an e-commerce site.

   Answer

   (a) **Availability**: the information has been neither stolen nor changed, but it cannot be reached. (b) **Confidentiality**: anyone who finds the stick can read the records; encryption would have prevented this risk. (c) **Integrity**: the information has been changed without permission.

10. In the rules of Table 1.4.25, if rule 4 (“all others, block”) is moved to the very top of the list, what happens to the connections 198.51.100.7 → 443 and 192.0.2.50 → 22?

Answer

Rules are tried from top to bottom and the first match is applied. Because the new first rule covers every connection, both connections are **blocked**; the rules that allow never get their turn. The website becomes unreachable from outside, and the administrator cannot connect to the server either. The default deny rule must always stay at the end of the list.

1.4.11 Further reading

- J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach* (8th ed., 2021): in the “top-down” order followed in these notes, it starts at the application layer and covers HTTP, DNS, TCP, IP routing, and the link layer in detail, with many figures.
- A. S. Tanenbaum, N. Feamster, and D. J. Wetherall, *Computer Networks* (6th ed., 2021): examines networks from the physical layer upward; the chapters on modulation and wireless transmission are especially useful for electrical and electronics students.
- R. Anderson, *Security Engineering* (3rd ed., 2020): explains, through many cases, why phishing, passwords, authentication, and encryption fail in real systems.
- Internet Society, *A Brief History of the Internet*: tells the development of the internet from ARPANET to today, briefly and readably, from the pens of the people who designed it.

1.4.12 References

1. Internet Society, *A Brief History of the Internet*, B. M. Leiner et al., 1997.
2. IETF, *Internet Standards and RFC documents*; especially RFC 1918 (private address blocks, 1996), RFC 5737 (IPv4 addresses used in documentation, 2010), RFC 8446 (TLS 1.3, 2018), and RFC 9293 (TCP, 2022).
3. W3C, *Web Standards*.
4. NIST, *Cybersecurity and Privacy guidance*; especially SP 800-63B, *Digital Identity Guidelines: Authentication and Lifecycle Management*.
5. Law No. 6698 on the Protection of Personal Data, Official Gazette, 7 April 2016, issue 29677.
6. TÜBİTAK ULAKBİM, *Annual Report*, 2001 and 2012 (the founding of ULAKBİM in 1996 and the operation of ULAKNET from 1997).
7. J. F. Kurose, K. W. Ross, *Computer Networking: A Top-Down Approach*, 8th ed., Pearson, 2021.

PART I · CHAPTER 1.5

Computation and Computational Thinking

A computer is often pictured as a fast calculator. Yet a search engine's result list, the route a map suggests, and a phone's face recognition are computation too: in each of them, data is processed according to definite rules. These notes first explain this broad notion of computation and what computer science is about. They then cover the short history of computation, the move from fixed-program machines to stored-program computers, and the theoretical foundation of that move, the Turing machine; a Turing machine adding 1 to a binary number is traced step by step. The two kinds of knowledge, "what is" and "how to", are compared through square-root finding and Euclid's algorithm. The last part shows four thinking tools that bring a large problem into a form a computer can solve, namely decomposition, pattern recognition, abstraction, and algorithm design, with examples.

LEARNING OUTCOMES

Computation
Explain computation as the processing of data according to rules; give examples of computation beyond arithmetic; identify the input, the process, the output, and the resources of a task.

Models of computation
Distinguish fixed-program from stored-program machines; trace the operation of a Turing machine step by step in terms of tape, head, state, and transition rules; list the limits of computers.

Knowledge and algorithms
Distinguish declarative from imperative knowledge; trace the square-root method and Euclid's algorithm with a trace table; justify why an algorithm stops.

Computational thinking
Apply decomposition, pattern recognition, abstraction, and algorithm design to a real problem; choose test cases made up of normal, boundary, and invalid inputs for an algorithm.

1.5.1 Computation

To most people the word “computation” recalls the four operations from school: addition, subtraction, multiplication, and division. In computer science the notion is much broader. Looking up a word in a dictionary, sorting a class list by surname, and deciding when a traffic light turns green are all computation. This part introduces the common structure of computation and the parts that make it possible.

1.5.1.1 What is computation?

Finding the average of a student’s four exam grades is one of the simplest examples of computation. We have the grades: 70, 85, 60, and 75. These numbers are the material to be processed. The rules to apply to it are clear as well: first the grades are added, then the total is divided by 4. Applying the rules gives a result: 72.5. Every example of computation has these three parts: the data to be processed, the rules to apply, and the result produced.

Definition 1.5.1

Computation is the transformation of data into a result by processing it according to definite rules. The data to be processed is called the **input**, the step-by-step application of the rules is called the **process**, and the result produced is called the **output**.

Two words in the definition are especially important. The first is “rules.” Computation is not done on a whim: when the same rules are applied to the same input, the same output comes out every time. Whoever computes the average today, or tomorrow, finds 72.5. The second is “processing.” Merely storing or displaying data is not computation; the data must be transformed according to a rule.

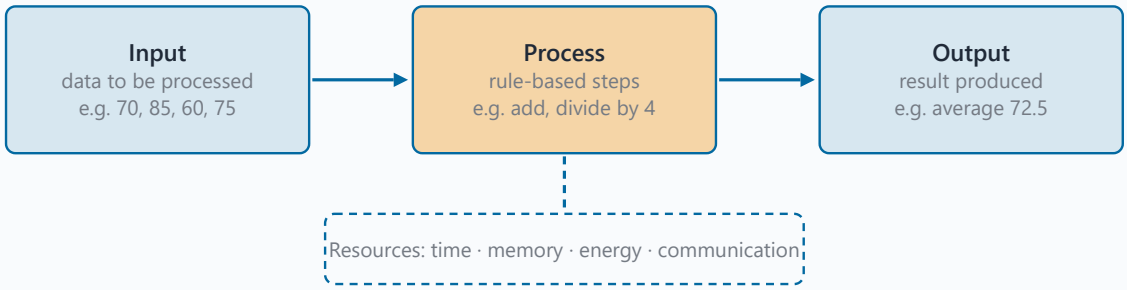


Figure 1.5.1: The three parts of a computation. The input (70, 85, 60, 75) is processed by rule-based steps (add, divide by 4) and the output (average 72.5) is produced. The process uses the resources time, memory, energy, and communication.

Figure 1.5.1 shows this structure. The resources in the dashed box are easy to overlook at first glance, but every computation depends on them. A process takes **time**: adding four numbers takes a moment, while sorting millions of records can take minutes. A process needs **memory**: the grades and the running total must be kept somewhere. Every step spends **energy**: the phone's battery and the data center's electricity bill are both consequences of this. If the data comes from somewhere else or goes somewhere else, **communication** is needed: the grades are read over the network from the student information system. Resources are not unlimited, so a method that does the same job with less time, memory, and energy is more valuable.

Example 1.5.1

How is the average of the grades 70, 85, 60, and 75 computed? Which intermediate result is kept at each step?

Step	Operation	Running total
1	The running total starts at 0	0
2	0 + 70	70
3	70 + 85	155
4	155 + 60	215
5	215 + 75	290
6	290 ÷ 4	72.5

Result: the average is 72.5. Addition was performed four times and division once. Memory holds only the total so far and the number of grades read. Even if the number of grades grew to 40, only these two values would be kept; what changes is the number of additions.

The last observation in the example is important. As the number of grades grows, the number of additions grows too; 40 grades need 40 additions. How the running time of a task grows with the size of its input is one of the criteria that decide whether a method is good.

1.5.1.2 Computation is not only arithmetic

Finding an average is a task done with numbers. Most of what computers do day to day, however, does not look directly related to numbers. An email program orders incoming messages by date and moves unwanted ones to a separate folder. A map application finds the shortest route between two points. A translation program renders a sentence in one language into another. All of these are processing of data according to rules, and therefore all of them are computation.

Kind	Input	Rule	Output
Search	A dictionary and the word "tape"	Find the word using alphabetical order	The meaning of the word

Kind	Input	Rule	Output
Sorting	Class list	Arrange the surnames in alphabetical order	Sorted list
Decision making	Exam grade 47	If the grade is below 50, “failed”; otherwise “passed”	“failed”
Transformation	Photograph	Turn the color of every pixel into gray	Black-and-white photograph
Communication	A message	Split the message into packets, address them, send them	Message that reaches the recipient

Table 1.5.1: Kinds of computation other than arithmetic.

Table 1.5.1 puts five different kinds of computation side by side. In none of them are the four operations in the foreground, yet all of them have the same three parts. In a search, the input is a dictionary and the word sought; the rule is to use alphabetical order to go to the right page. In decision making the rule is a comparison: since $47 < 50$ is true, the output is “failed.” In a transformation, every pixel of a photograph is changed by the same rule. Even communication has rules: a message is split into pieces of a certain size and an address is written at the head of each piece.

This breadth also explains why computers are used in so many fields. At its core a computer is a machine that processes data according to rules. If a task can be described with explicit rules and its data can be turned into numbers, the task can be given to a computer. Text, images, and sound are already stored in a computer as sequences of numbers, so operations on them are computation as well.

Example 1.5.2

A campus canteen deducts the lunch fee from the balance on a student card when the card is tapped. The fee is 45 TL and the balance on the card is 120 TL. Which kinds of computation are involved?

1. **Communication.** The card reader sends the card number to the canteen’s server.
2. **Search.** The server looks the card number up among the records and finds the balance: 120 TL.
3. **Decision making.** Since $120 \geq 45$ is true, the payment is accepted. If the balance were below 45 TL, the payment would be rejected.
4. **Arithmetic.** The new balance is computed: $120 - 45 = 75$ TL.
5. **Communication.** The result is sent back to the reader and the screen shows “Remaining: 75 TL.”

In a single card tap, four different kinds of computation run one after another. Arithmetic is only one of them.

Common mistake

Thinking that only electronic computers carry out computation. Computation is a process, not a machine. A teacher sorting a class list with pen and paper, and a driver comparing routes and choosing the shortest, are computing too. A computer is a tool that carries out the same process much faster and without mistakes.

1.5.1.3 The building blocks of computation

For a computation to be designed on paper and run on a machine, several different parts have to come together. These parts can be grouped under six headings.

Mathematics
: Makes it possible to operate on quantities and relations. Operations such as addition, division, and comparison are the tools of mathematics.

Logic
: Decides whether a condition is true or false. The answer to "Is the grade below 50?" is a logical value: true or false.

Data
: The inputs that represent the problem and the intermediate results produced during the computation. In the average example, the grades and the running total are data.

Algorithm
: A finite, orderly sequence of operations. "Set the total to 0, add each grade, divide by the number of grades at the end" is an algorithm.

Hardware
: The system that physically carries out the steps: processor, memory, wires. However good an algorithm is, no result is obtained without a machine to run it.

Programming language
: The means of writing the algorithm in a form the computer can understand. The same algorithm can be written in Java, Python, or C.

These parts do not replace one another; they complement one another. Mathematics and logic determine which operations can be performed, data determines what is processed, and the algorithm determines the order of operations. The programming language puts the algorithm into a form the machine understands, and the hardware runs it. A recipe analogy makes the relationship concrete: the ingredients are data, the recipe is the algorithm, the language the recipe is written in is the programming language, and the stove and pot are the hardware. Measurements and checks such as "Is it thick enough yet?" are the share of mathematics and logic.

Building block	Counterpart in the average computation
Mathematics	Addition and division: *(70 + 85 + 60 + 75) ÷ 4*
Logic	The question "Is there a grade left to read?"; when the answer is false, the addition ends

Building block	Counterpart in the average computation
Data	The four grades, the running total, and the number of grades
Algorithm	Set the total to 0; add each grade to the total; divide the total by the number of grades; write the result
Hardware	The processor that keeps the total in memory and does the addition
Programming language	The algorithm written in a language such as Java

Table 1.5.2: The building blocks in the average computation.

Common mistake

Thinking that an algorithm and a program are the same thing. An algorithm is the idea that describes the steps by which a task is done; it can be written in English sentences or as a flowchart. A program is the counterpart of that idea in a particular programming language. One algorithm can have many different programs. If the algorithm is wrong, the program gives wrong results whatever language it is written in.

1.5.2 Computer science

Knowing how to use a computer and knowing computer science are different things. A person who prepares a document in a word processor is using a computer. Computer science asks questions such as how that word processor stores the text, by what method the spell checker works, and why the program slows down as the document grows. This part introduces the core areas of interest of computer science and the responsibilities of the field toward society.

1.5.2.1 What does computer science study?

Definition 1.5.2

Computer science is the discipline that studies how information is represented, how it is processed, and by what methods problems can be solved. It concerns not only using tools but also the methods behind the tools and the limits of those methods.

The broad field of computer science can be grouped under four main areas of interest.

Abstraction. Hiding unnecessary details and focusing on the essential structure. A user who saves a file does not know which magnetic region of the disk or which memory cell the data is written to; they see only the “Save” button. The details of the disk are hidden behind a layer of abstraction. Computer systems consist of many such layers stacked on top of one another. Abstraction is also one of the components of computational thinking.

Algorithms. Designing solution steps and analyzing them. Analysis asks whether an algorithm gives the correct result and how much time and memory it uses. Of two algorithms that solve the same problem, one may finish in a second and the other in an hour.

Systems. Understanding how hardware and software work together. Operating systems, networks, databases, and compilers are topics under this heading. Why a program runs slowly often depends not on the algorithm but on some part of the system, for example waiting for data to arrive over the network.

People and society. Evaluating the effects of technology on people, its security, and its ethics. A piece of software can work flawlessly from a technical point of view and still harm people.

Example 1.5.3

How does a phone's navigation application touch the four areas of interest of computer science?

1. **Abstraction.** The city is kept as a network of intersections and the roads between them. The colors of buildings, trees, and signs do not enter the model. For each road only its length and average travel time are stored.
2. **Algorithms.** An algorithm runs that finds the route with the shortest travel time between two points. In a network with thousands of intersections, trying every route one by one would take far too long, so a method that narrows the search cleverly is needed.
3. **Systems.** The location comes from the GPS receiver, the traffic information is read from a server over the network, and the map is drawn on the screen. The phone's processor, memory, network connection, and operating system work together.
4. **People and society.** The application knows where the user is. With whom this location data is shared is a question of privacy. Whether a user who cannot see the screen can receive spoken directions is a question of accessibility.

1.5.2.2 Technology and responsibility

Computers are now used in decisions that directly affect human lives, from banking to health, from education to hiring. A piece of software therefore has to do more than work correctly: it must also be fair, secure, and usable by everyone. Four areas of responsibility stand out.

Accessibility

Equal access to technology for different users. A visually impaired user must be able to listen to a web page with a screen reader. For this, every image on the page needs a text description. A color-blind user cannot read a chart separated only by red and green; the information must be given by shape or text in addition to color.

Privacy

Using personal data only for its stated purpose and securely. A shopping app asks for an address for delivery; selling the same address to advertisers without the user's knowledge goes beyond that purpose. In Turkey, Law No. 6698 on the Protection of Personal Data sets rules on this subject.

Fairness

Data and algorithms must not produce discriminatory results. If an algorithm learns from past data, it can learn past biases as well. Even if the algorithm itself looks neutral, its results can work against particular groups.

Sustainability

Taking energy consumption, hardware lifetime, and electronic waste into account. Data centers use large amounts of electricity. Devices that become obsolete quickly turn into electronic waste, and this waste contains metals that harm nature.

Example 1.5.4

A bank uses software that evaluates loan applications. The software gives a score by looking at the applicant's income, debt history, and neighborhood. During an audit it is noticed that applications from certain neighborhoods receive lower scores even when income and debt history are similar. Which area of responsibility does this concern, and what can be done?

1. **Identify the area.** People in similar situations get different results only because of where they live. This is a discriminatory outcome; the subject is *fairness*.
2. **Look for the cause.** The neighborhood may be correlated with income level or other social characteristics. The algorithm may be drawing an indirect distinction by using the neighborhood.
3. **Remedy.** The neighborhood information can be removed from the scoring. Results are compared across groups at regular intervals. A person whose application is rejected is given a way to learn the reason and to appeal.

The same system also processes the applicants' identity and income information. Using this information only for the loan evaluation and storing it securely is a privacy responsibility. A single piece of software can involve more than one area of responsibility at the same time.

Common mistake

Thinking "if a computer decided, it is impartial." An algorithm is influenced by the data it is given and by the choices of those who design it. If the data it was trained on is biased, its results are biased too. Impartiality does not come by itself; it must be measured and audited.

1.5.3 The history of computation

Today's computers are the product of ideas accumulated over several generations. Many of these ideas were conceived long before the technology to implement them appeared. This part describes the main milestones in the history of computation and, perhaps the most important of them, the idea of the stored program.

1.5.3.1 Milestones

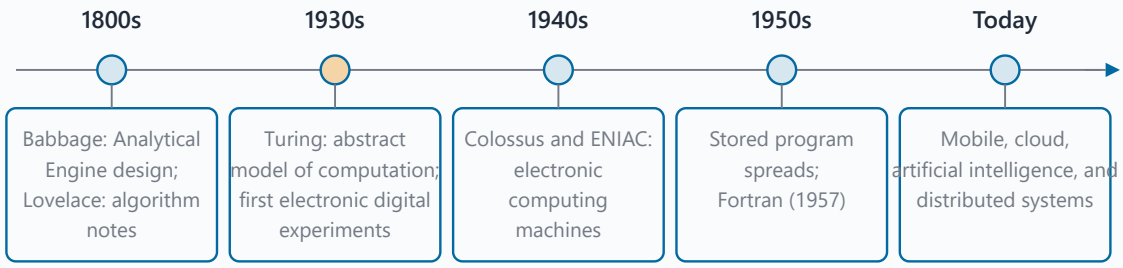


Figure 1.5.2: The five eras in the history of computation. The 1930s are highlighted as the period in which the theoretical foundation of computation was laid.

Figure 1.5.2 shows the five eras on a timeline. Each era takes the ideas of the one before and implements them with faster, smaller, and cheaper technology.

The 1800s: mechanical designs. The English mathematician Charles Babbage designed a mechanical computing machine in the 1830s, which he called the Analytical Engine. The machine was to run on gears. It would have a separate computing unit and a memory unit that held numbers. It would also read its instructions from punched cards. The idea of punched cards came from the Jacquard loom, whose patterns were determined by holes in cards. Babbage's machine could not be completed with the craftsmanship available at the time. Even so, its design for the first time combined the basic parts of today's computers. In her 1843 notes on the machine, the mathematician Ada Lovelace gave a step-by-step method for computing Bernoulli numbers. This method is regarded as one of the first algorithms written to be run on a machine. Lovelace foresaw that the machine could operate not only on numbers but on anything that can be expressed with symbols, for example musical notes.

The 1930s: theoretical foundation. In a paper published in 1936, Alan Turing defined an abstract model of computation. Known today as the Turing machine, this model gave a mathematical answer to the question "what exactly is a computation?" The same paper also showed that some problems cannot be solved by any machine. In these years the first experimental machines that computed with electronic circuits also began to be built.

The 1940s: electronic machines. During the Second World War, the Colossus machines were built in Britain to break encrypted messages. In the United States, ENIAC was developed to compute firing tables for artillery shells. These machines ran on thousands of vacuum tubes and filled a whole room. They were thousands of times faster than mechanical machines. However, to move on to a new job, the wires and switches had to be rearranged by hand.

The 1950s: the stored program and the first high-level languages. Computers that also kept their instructions in memory became widespread. Programmers were now dealing not with the machine's wires but with instructions in memory. Fortran, developed by IBM in 1957, made it possible to write scientific calculations in a language resembling mathematical formulas. Fortran became the first widely used high-level programming language.

Today: computation everywhere. Thanks to the transistor, the integrated circuit, and the microprocessor, computers have shrunk to fit in a pocket. Today phones, cloud services, artificial intelligence applications, and distributed systems made up of thousands of connected machines all run on the same basic ideas.

The lesson of this short history is this: as hardware became faster and smaller, basic ideas such as the algorithm and the stored program became applicable in wider fields. The ideas did not change; the technology carrying them did. Babbage's gears, ENIAC's vacuum tubes, and today's transistors do the same job: they process data according to rules.

Era	Technology	How is the program supplied?
1800s	Gears, levers	Punched cards (design)
1930s	Abstract model on paper	Table of transition rules
1940s	Vacuum tubes	Wiring and switch layout
1950s	Vacuum tubes, then transistors	Instructions stored in memory; Fortran
Today	Chips with billions of transistors	Programs written in high-level languages

Table 1.5.3: The technology carrying computation in each era and how the program is supplied.

Common mistake

Thinking that one person invented the computer at a single moment. The computer developed over more than a hundred years with the contributions of many people in different countries. Babbage put forward the design, Lovelace the idea of the algorithm, Turing the theoretical model, and the engineers of the war years the electronic machines. Each built on the one before.

1.5.3.2 Fixed program and stored program

The great majority of early machines were built to do a single job. A desk calculator performs only the four operations. The rotary knob of an old washing machine selects one of a few fixed programs determined by gears. In these machines the “program” is the physical structure of the machine itself. To make the machine do a new job, it has to be rebuilt.

Definition 1.5.3

In a **fixed-program** machine, the task to be done is determined by the physical layout of the machine or by a special mechanism. In a **stored-program** machine, the instructions are kept in memory, just like the data to be processed. The machine reads the instructions from memory one by one and carries them out.

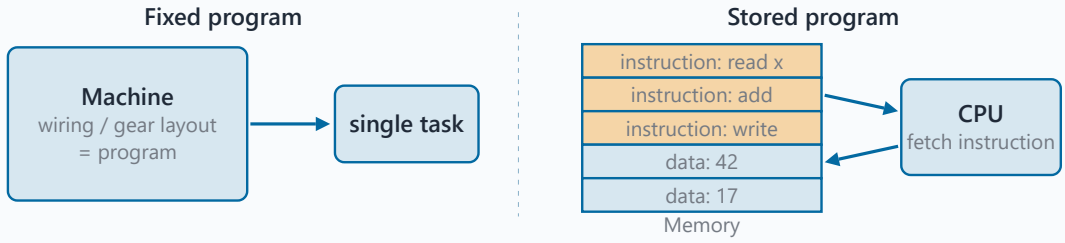


Figure 1.5.3: Left, a fixed-program machine: the wiring or gear layout is the program itself, and the machine does a single task. Right, a stored-program machine: the instructions (read x, add, write) and the data (42, 17) sit in the same memory; the central processing unit (CPU) fetches the instructions in order.

Figure 1.5.3 puts the two approaches side by side. The memory on the right has five cells. The top three hold instructions and the bottom two hold data. As far as memory is concerned, there is no difference between these cells; they are all bit sequences. What makes the content of a cell an instruction is that the processor reads it as an instruction. The central processing unit (CPU) fetches an instruction from memory, decodes and executes it, then moves on to the next instruction. The instruction reads the data it needs from the same memory.

The consequence of this simple idea is large. To make the machine do a new job, there is no need to touch the wires; loading new instructions into memory is enough. The same computer becomes a word processor in the morning, a game in the afternoon, and a video player in the evening. Physically nothing changes; only the instructions in memory change. On ENIAC, switching to a new job could take days because the wires had to be reconnected. On a stored-program machine, the same switch takes only as long as loading a new program into memory.

The stored-program idea spread to a wide audience through the report John von Neumann wrote in 1945 for a machine called EDVAC. This structure, in which instructions and data sit in the same memory, is today called the **von Neumann architecture**. The first electronic computer to run a

program stored in memory was a small experimental machine that ran at the University of Manchester in England in 1948. Almost all of today's computers are a variant of this architecture.

Approach	How is the program determined?	Flexibility
Fixed program	The physical layout or a special mechanism of the machine	A specific task
Stored program	Instructions are stored in memory like data	Different tasks by changing the program
General purpose	An architecture that can run very different programs	A wide range of applications

Table 1.5.4: Comparison of fixed-program, stored-program, and general-purpose machines.

Table 1.5.4 also contains a third row: the **general-purpose** computer. The stored-program idea is a precondition for the general-purpose computer. A machine whose instructions can be changed can carry out almost any computation with suitable instructions. Personal computers and smartphones are general purpose. The opposite is special-purpose devices designed for a single job.

Example 1.5.5

Does each of the following three devices work with a fixed program or a stored program? A mechanical wristwatch, a game console, and a digital camera.

1. **Mechanical wristwatch.** What the watch does is determined by the ratios of its gears. To make the watch do a new job, the gears must be changed. It is fixed-program.
2. **Game console.** Each new game is a new program loaded into memory. The same hardware runs thousands of different games and gains new features through updates. It is stored-program.
3. **Digital camera.** To the user it seems to do a single job: taking photographs. Inside, however, there is a processor running software stored in memory. The manufacturer can add new shooting modes to the camera with a software update. Seen from inside it is stored-program; but because the user cannot load their own programs, it is not general purpose.

The third device shows that the two distinctions are independent of each other. A device can work with a stored program and still be designed for a single job. So are the small computers inside today's washing machines, ovens, and cars.

Common mistake

Thinking that a stored-program machine is a "faster" machine or one with "more memory." The distinguishing feature is not speed but where the instructions are kept. If the instructions are stored in memory like data and can be changed, the machine is stored-program. A very

fast machine can be fixed-program, and a slow machine can be stored-program.

1.5.4 Models of computation

To ask what a computer can and cannot compute, we first need a precise definition of "computation." Real computers are too complicated for this: processor types, memory sizes, and programming languages differ greatly. Computer science therefore uses simple machine models stripped of details. The most famous of these models is the Turing machine. This part introduces the parts of a Turing machine, runs a machine step by step, and explains what this model says about the limits of computers.

1.5.4.1 The Turing machine

Turing built his model by observing a person doing a calculation with pen and paper. The person writes symbols on the paper, reads what has been written, and decides what to do in the next step by looking at the information held in mind at that moment and the symbol being seen. Turing broke this behavior down into its simplest parts.

Definition 1.5.4

A **Turing machine** is an abstract machine model used to explain algorithmic computation. It consists of four parts:

- **Tape**: a strip divided into cells that can extend as far as needed in both directions. Each cell holds a **symbol**. A blank cell is shown as □.
- **Read/write head**: at any moment it is over a single cell. It reads the symbol in that cell, may write a new symbol in its place, and then moves one cell right or left, or stays where it is.
- **State**: the information the machine is "holding in mind" at that moment. States are written with names such as q0 and q1. When the machine reaches the special state HALT, it finishes running.
- **Transition rules**: a table that says what to do for each (state, symbol read) pair. A rule specifies three things: the symbol to write, the movement of the head, and the new state.

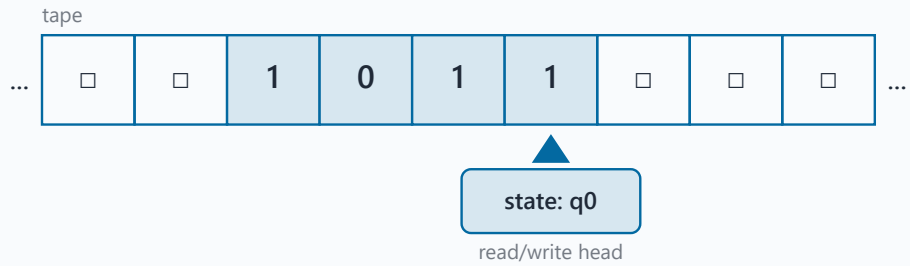


Figure 1.5.4: A Turing machine. The tape reads □□1011□□□; the three dots on either side show that the tape continues in both directions. The read/write head is over the rightmost 1 (cell 6) and the machine is in state q0.

Figure 1.5.4 is a snapshot of the machine. The binary number 1011 is written on the tape; the remaining cells are blank. The triangle under the head marks the cell to be read. The box under the head shows the machine's state: q0.

The power of the model comes from its simplicity. A step of the machine always has the same form:

1. The symbol under the head is read.
2. In the transition-rule table, the row matching the (current state, symbol read) pair is found.
3. The symbol the row specifies is written into the cell.
4. The head moves one cell in the direction the row specifies, or stays in place.
5. The machine goes to the new state the row specifies. If the new state is HALT, the machine stops; otherwise it returns to the first step.

The machine's "intelligence" is only in the rule table. The machine looks at the table and does what it says; it does not remember past steps and does not plan ahead. If there is something it needs to remember, it carries it either as a symbol on the tape or in its state. The tape of unbounded length is the machine's memory. Real computers have finite memory; the model sets the memory problem aside and focuses only on the question "can this job be done with rules?"

Common mistake

Thinking that a Turing machine is a device that was actually built. A Turing machine is a model defined on paper; its purpose is not to build a computer but to define precisely what computation is. Enthusiasts have since built small demonstration machines, but the value of the model is theoretical.

1.5.4.2 Transition rules

Only the rule table determines what a Turing machine does. The first example is a machine that adds 1 to a binary number written on the tape. Adding 1 in binary is similar to adding 1 in decimal. We start at the rightmost digit. If the digit is 0, it becomes 1 and the job is done. If the digit is 1, it

becomes 0 and a carry is passed to the digit on its left; the same operation is repeated there. If no digit is left to the left of the number, a 1 is written there.

Rule	State	Read	Write	Move	New state	Meaning
1	q0	1	0	left	q0	1 + 1 = 10: the digit becomes 0, the carry moves left
2	q0	0	1	halt	HALT	0 + 1 = 1: the carry ends at this digit
3	q0	□	1	halt	HALT	Left of the number is blank: the carry becomes a new digit

Table 1.5.5: Transition rules of the Turing machine that adds 1 to a binary number. The start state is q0; the head starts at the rightmost digit of the number.

Table 1.5.5 consists of three rules. The machine has a single working state, q0. Three symbols can be read in this state: 1, 0, and □. For each of them there is exactly one rule in the table. The machine is therefore never undecided about what to do at any step. Rule 1 does the carrying and leaves the machine in q0, so the same rule can be applied to all consecutive 1s. Rules 2 and 3 stop the machine.

This table fits the knowledge of binary addition into just three rows. The machine does not know the notions of "binary number" or "carry"; it only reads and writes symbols. It is the human who designed the table who gives the rules their meaning.

1.5.4.3 Step by step: 111 + 1

Let us run the machine with 111 written on the tape. In decimal, 111_2 is *4 + 2 + 1 = 7*; the expected result is 8, that is, 1000_2. The head is over the rightmost digit and the state is q0.

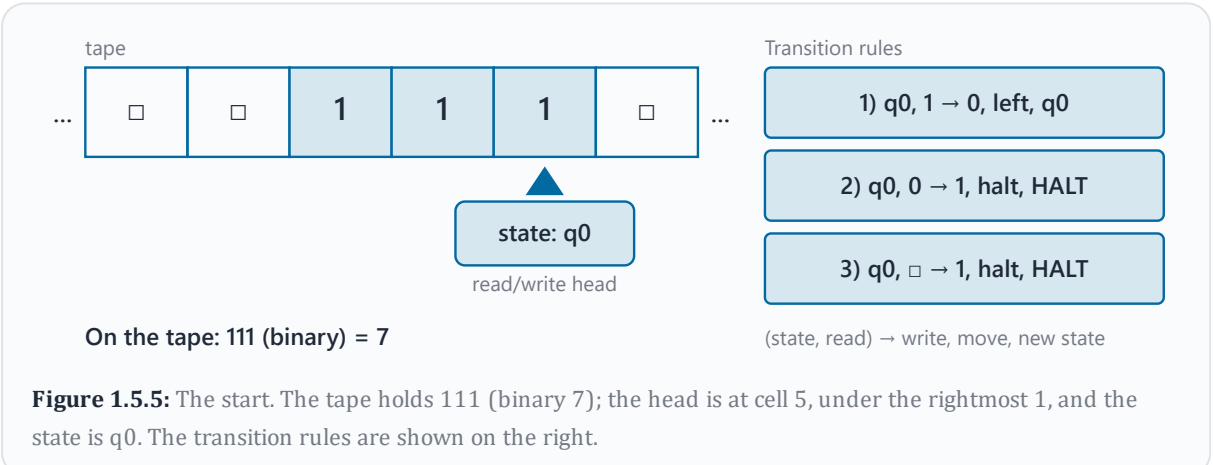


Figure 1.5.5: The start. The tape holds 111 (binary 7); the head is at cell 5, under the rightmost 1, and the state is q0. The transition rules are shown on the right.

From the starting situation in Figure 1.5.5, the machine follows these steps:

1. **Step 1.** The symbol read is 1, the state is q0. The matching rule is rule 1: 0 is written, the head moves left, and the state stays q0. The tape becomes □□110□. At the rightmost digit, 1 + 1 = 10 has been computed: 0 was written in the digit and the carry 1 was moved one cell to the left.
2. **Step 2.** The head is at cell 4 and the symbol read is again 1. Rule 1 is applied again. The tape becomes □□100□; the carry is still going.

3. **Step 3.** The head is at cell 3 and the symbol read is 1. Rule 1 is applied a third time. The tape becomes □□000□. All the digits of the number have turned to 0, but the carry has not been written yet.
4. **Step 4.** The head is at cell 2 and the symbol read is □. The matching rule is rule 3: 1 is written and the machine goes to the state HALT. The tape becomes □1000□.

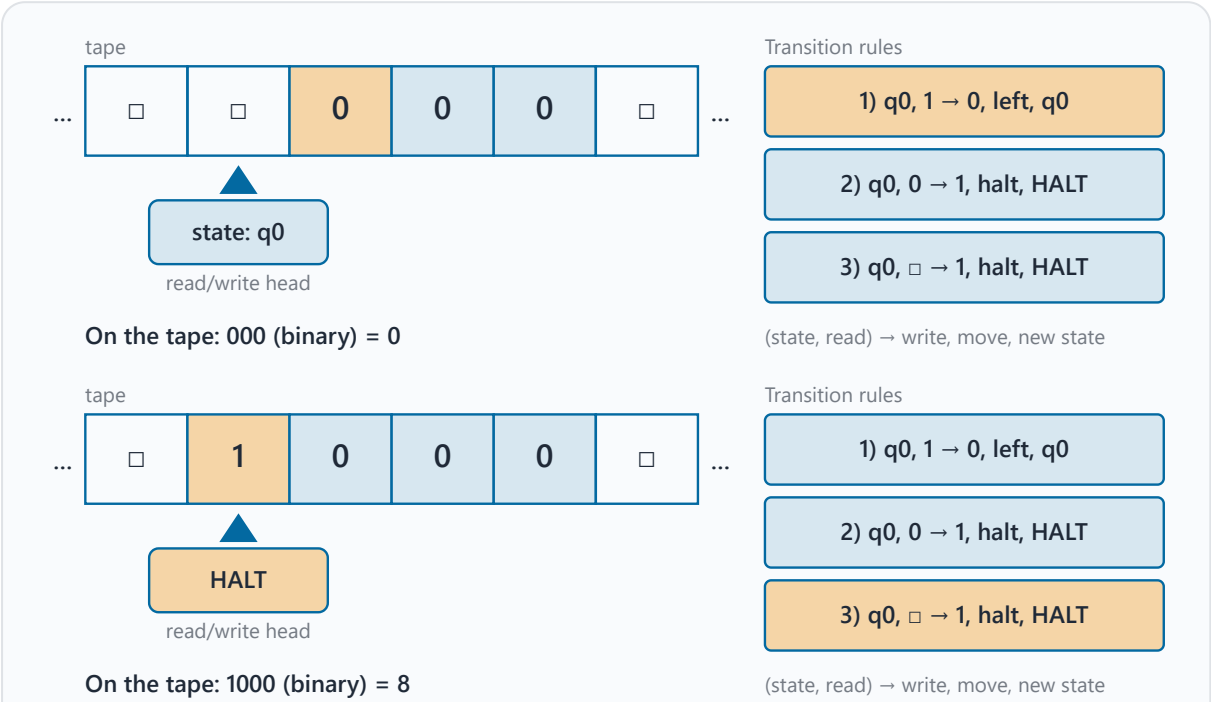


Figure 1.5.6: The last two steps. (a) After rule 1 has been applied three times, the tape is □□000□; the head is about to read the blank cell at cell 2. (b) Rule 3 has written a 1 and the machine has halted: the tape is □1000□, binary 8.

Figure 1.5.6 shows the last two steps. The whole run is summarized in the trace table below. Each row of the table shows the situation before a rule is applied and the tape after it is applied.

Step	Head	State	Read	Rule	Tape after	Head after	State after
1	5	q0	1	1	□□110□	4	q0
2	4	q0	1	1	□□100□	3	q0
3	3	q0	1	1	□□000□	2	q0
4	2	q0	□	3	□1000□	2	HALT

Table 1.5.6: Trace table for 111 + 1. The cells are numbered from 1, starting at the left.

The machine halted in four steps and 1000 was left on the tape: $1000_2 = 8 = 7 + 1$. The result is correct. Table 1.5.6 also contains an important observation: in every row, only the “State” and “Read” columns determined what was to be done. The machine’s earlier steps, the rest of the tape, and how many digits the number had did not influence any decision.

The number of steps also depends on the input. The machine spends one step for each run of consecutive 1s from the right, and one more step where the carry ends. For a number with k consecutive 1s at its right end, the machine halts in *k + 1* steps. 111 has three consecutive 1s at its right end; the number of steps is *3 + 1 = 4*.

Example 1.5.6

The same machine is run with 10011 (binary 19) on the tape; the head is at the rightmost digit. In how many steps does the machine halt, and what is left on the tape?

Step	Read	Rule	Action	Tape after
1	1	1	write 0, move left	10010
2	1	1	write 0, move left	10000
3	0	2	write 1, HALT	10100

The machine halts in three steps. 10100 is left on the tape: *16 + 4 = 20 = 19 + 1*. Since there are two 1s at the right, the number of steps is *2 + 1 = 3*. The digits to the left of the 0 where the carry ended (the leftmost 1 and the 0) were never touched. Rule 3 was not used at all on this input, because the carry ended before the head reached a blank cell.

Common mistake

Thinking that the machine stops after applying rule 1. The new state of rule 1 is q0, not HALT; the machine keeps running. The machine stops only when a rule whose new state is HALT is applied. In this machine those are rules 2 and 3.

1.5.4.4 Memory in the state: a parity check

The machine that adds 1 had a single working state. For some tasks the machine must keep in mind some information about the symbols it has read. This information is stored in the state. A machine that finds out whether the number of 1s in a bit string is even or odd shows this clearly. This machine, one of the ready-made machines in the Turing Machine simulator, has two working states: “even” and “odd.” The head starts at the leftmost symbol of the string and moves right.

Rule	State	Read	Write	Move	New state	Meaning
1	even	0	0	right	even	0 does not change the number of 1s
2	even	1	1	right	odd	One more 1: the number became odd
3	odd	0	0	right	odd	0 does not change the number of 1s
4	odd	1	1	right	even	One more 1: the number became even
5	even	□	E	halt	HALT	End of the string; the number of 1s is even

Rule	State	Read	Write	Move	New state	Meaning
6	odd	□	0	halt	HALT	End of the string; the number of 1s is odd

Table 1.5.7: Transition rules of the Turing machine that performs a parity check. The start state is “even.”

The machine in Table 1.5.7 moves along the tape without changing anything; it writes back into each cell the symbol it read. The only thing that changes is the state. When the machine reads a 1, the state flips from even to odd or from odd to even; when it reads a 0, the state stays the same. When the blank cell at the end of the string is reached, the state says whether the number of 1s read so far is even or odd. The machine writes this answer into the blank cell as E or O and halts.

Example 1.5.7

The parity machine is run with 1101 on the tape. How does the state change at each step, and what is written on the tape?

Step	Head	State	Read	Rule	New state
1	1	even	1	2	odd
2	2	odd	1	4	even
3	3	even	0	1	even
4	4	even	1	2	odd
5	5	odd	□	6	HALT (O written)

The machine halts in five steps and the tape becomes 1101O. The string has three 1s; three is odd. Without doing any counting, only by moving back and forth between two states, the machine reached the right answer. The number of steps is one more than the length of the string: four symbols and one blank cell.

This machine shows that the state of a Turing machine is a kind of small memory. The machine does not know the exact number of 1s; it knows only as much as the task requires, namely whether the number is even or odd. The same idea is used in real computers: the parity bit sent to detect whether data was corrupted in transmission is found with exactly this computation.

1.5.4.5 The universal Turing machine

Every Turing machine seen so far did a single job: one added 1 to a number, the other checked parity. A new job needed a new rule table, that is, a new machine. In this respect every Turing machine resembles a fixed-program machine. In the same paper, Turing also defined a machine that goes beyond this limit.

Definition 1.5.5

A **universal Turing machine** is a Turing machine that, when it starts running, has two things written on its tape: the rule table of another Turing machine and the input of that machine. The universal machine reads this table and applies it step by step, doing exactly the job that machine would do; that is, it imitates it.

The universal machine's own rule table is fixed and never changes. What changes is the table written on its tape. If the table of the machine that adds 1 is written, it adds 1; if the table of the parity machine is written, it checks parity. The rule table is no longer part of the machine but data sitting on the tape.

This idea is the theoretical foundation of the stored-program idea defined in Definition 1.5.3. The tape of the universal Turing machine corresponds to the computer's memory, the rule table on the tape to the program in memory, and the fixed rules of the universal machine to the processor's circuits that decode and carry out instructions. Turing put this idea forward on paper in 1936. Real stored-program computers could be built only more than ten years later.

Universal Turing machine	Stored-program computer
Tape	Memory
Rule table written on the tape	Program in memory
Input written on the tape	Data in memory
Fixed rules of the universal machine	The processor circuits that fetch and carry out instructions

Table 1.5.8: Correspondences between the universal Turing machine and a stored-program computer.

Although the Turing machine looks simple, it is very powerful. Given enough time and memory, a Turing machine can carry out every computation that any computer known today can. In other words, no computer today can perform a computation that a Turing machine cannot. Computers are much faster than Turing machines, but the set of things they can compute is the same. The Turing machine is therefore an adequate model for studying the question "what can a computer compute?"

Common mistake

Thinking that the universal Turing machine is the one with the infinite tape or the two-way head. The infinite tape and a head that can move in both directions are in every Turing machine. What sets the universal machine apart is that it takes the description of another machine as input and can imitate it.

1.5.4.6 The limits of computers

Computers are very fast, but they cannot do everything. Some limits come from resources, some from the nature of data and models, and some from computation itself. Six limits stand out.

Time
: Large inputs can make some algorithms impractical. If an algorithm's running time grows very quickly as the input grows, the result does not arrive within a time the user can wait.

Memory
: Data and intermediate results are kept in a finite storage area. The tape of a Turing machine is infinite; the memory of a real computer is not.

Energy
: Every operation and every data transfer consumes energy. A phone's battery and a data center's cooling system and electricity bill are the everyday faces of this limit.

Computability
: For some problems there is no algorithm that gives the correct result on every input. The best-known example is the halting problem.

Data quality
: Missing or faulty input leads even a correct algorithm to a wrong result.

Model limits
: A program represents only selected features of the real world. A feature that is not in the model is not taken into account in the result either.

The first three limits are resource limits. A faster processor, a larger memory, or a more efficient algorithm pushes these limits back. Computability is a different kind of limit: it cannot be overcome however powerful hardware becomes. The last two limits arise from the connection of computation to the real world.

Example 1.5.8

A program tries to break a password by trying all possible keys one by one, and it can try 10^9 (one billion) keys per second. If the key is 40 bits long, how long does it take in the worst case? What happens if the key is 64 bits?

1. **Number of possible keys.** With n bits, 2^n different keys can be written. For 40 bits there are $2^{40} = 1{,}099{,}511{,}627{,}776$, about 1.1×10^{12} keys.
2. **Time for 40 bits.** $2^{40} \div 10^9 \approx 1{,}099.5$ seconds, about 18 minutes.
3. **Time for 64 bits.** There are $2^{64} \approx 1.84 \times 10^{19}$ keys. $1.84 \times 10^{19} \div 10^9 \approx 1.84 \times 10^{10}$ seconds. Since a year is about 3.16×10^7 seconds, the time is about 584 years.
4. **Effect of hardware.** A computer twice as fast halves the time, to about 292 years. Adding just one bit to the key doubles the time again.

When the key length grows from 40 bits to 64 bits, an increase of only 24 bits, the time grows by a factor of $2^{24} \approx 16.8$ million. Algorithms whose running time doubles with every bit of input in this way become impractical on large inputs. Encryption systems rely on exactly this time limit.

The halting problem. The best-known example of the computability limit is the **halting problem**. The question is this: given any program and its input, can a general algorithm be written that correctly decides in every case whether the program will ever stop? For some programs the answer is easy. It is clear that the Turing machine that adds 1 halts on every input. For others, finding the answer is very hard. The short program below is a famous example:

```
n = 6
while n ≠ 1
    if n mod 2 = 0
        n = n / 2
    else
        n = 3 * n + 1
print("stopped")
```

If the program is run with n = 6, n takes the values 6, 3, 10, 5, 16, 8, 4, 2, 1 in turn and the program stops after eight iterations. If it is run with n = 27, it takes 111 iterations and the intermediate values rise as high as 9,232. Whether this program stops for every positive n is still unknown today. It has been tried with computers up to very large numbers and has always been seen to stop, but it has not been proved that it stops for every n.

In 1936 Turing showed something stronger: a general algorithm that correctly decides, for every program and every input, whether it will stop can *never* be written. The main idea of the argument is as follows. Suppose such an "oracle" algorithm exists. Using this oracle, a program that behaves contrarily can be written: the program asks the oracle about itself; if the oracle says "it halts," the program loops forever, and if the oracle says "it does not halt," the program halts at once. When this contrary program is run on itself, the oracle is wrong whatever it answers. So such an oracle cannot exist. This result has nothing to do with speed or memory; it is a logical impossibility.

Example 1.5.9

A weather station sends four temperature readings within an hour: 21.4 °C, 22.1 °C, 23 °C, and −99 °C. When the sensor cannot take a reading, it sends the value −99. What is the result if the program that computes the average uses all four values?

1. **Sum.** *21.4 + 22.1 + 23 + (−99) = −32.5*.
2. **Average.** *−32.5 ÷ 4 = −8.125* °C.

3. **Correct computation.** −99 is not a measurement but a marker meaning “no reading”; it must be removed from the computation. The average of the remaining three values is $(21.4 + 22.1 + 23) \div 3 = 66.5 \div 3 \approx 22.17$ °C.

The averaging algorithm worked flawlessly; the error was in the data. This situation is called “garbage in, garbage out.” The data-quality limit can be overcome only by checking the data beforehand.

Model limits. A navigation application models roads by their lengths and average speeds. If the model does not know that a road was closed that morning because of flooding, the application keeps recommending the closed road. The program did not compute wrongly; it could not know a feature of the real world that is missing from the model. Every program works with a model, and everything outside the model is outside the program as well.

Common mistake

Thinking that every limit can be overcome with more powerful hardware. The limits of time, memory, and energy can be eased with better hardware or a better algorithm. Computability limits such as the halting problem, however, are theoretical: such an algorithm does not exist at all, so no hardware can run it.

1.5.5 Knowledge and algorithms

There are two different ways of knowing something. A person can know a fact: “Ankara is the capital of Turkey.” A person can also know how something is done: “To go to Ankara, take this bus and get off at this stop.” Computer science is mainly concerned with the second kind, because to make a computer do a job, you have to describe how the job is done, step by step. This part compares the two kinds of knowledge, defines the algorithm, and traces two classic algorithms in detail: square-root finding and Euclid’s algorithm.

1.5.5.1 Declarative and imperative knowledge

Definition 1.5.6

Declarative knowledge explains what is true or what a concept is. **Imperative knowledge** explains, in steps, how to reach a result.

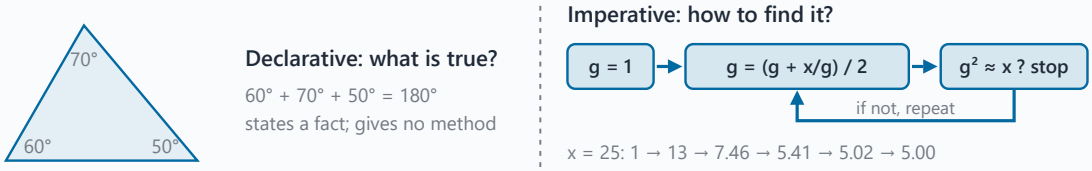


Figure 1.5.7: The two kinds of knowledge. Left, declarative knowledge: the interior angles of a triangle are 60°, 70°, and 50°, and their sum is 180°; this is a fact and gives no method. Right, imperative knowledge: to find a square root, start with a guess, update the guess with a formula, and stop when the square of the guess is close enough to x. For x = 25 the guesses are 1, 13, 7.46, 5.41, 5.02, and 5.00.

Figure 1.5.7 puts the two kinds side by side. The sentence “the interior angles of a triangle add up to 180°” gives a true piece of information but does not tell us how to do a job. With this knowledge the missing angle of a triangle can be found; but even for that a method is still needed. The steps on the right, by contrast, can be applied directly. A person or machine that follows the steps reaches the result even without fully understanding what a square root is.

The definition of the square root is also declarative knowledge: “the square root of x is the positive number whose square is x.” This definition is useful for checking whether a number is a square root. Since the square of 5 is 25, 5 is the square root of 25. But the definition does not show how to find the square root of 10. Declarative knowledge answers the question “what,” imperative knowledge the question “how.”

Topic	Declarative knowledge (what?)	Imperative knowledge (how?)
Square root	$\sqrt{x}$ is the positive number whose square is x.	Start with a guess, update the guess with the averaging formula, stop when it is close enough.
Prime number	A prime number is an integer greater than 1 that is divisible only by 1 and itself.	Divide the number by every number from 2 up to its square root; if none divides it evenly, the number is prime.
Sorted list	Each element is not greater than the one after it.	Find the smallest element and put it first; repeat the same step with the remaining elements.
Directions	The library is north of the main building.	Enter through the main gate, turn right, and walk to the second building.

Table 1.5.9: Examples of declarative and imperative knowledge.

Table 1.5.9 shows the two faces of the same topic in each row. The sorted-list row is especially instructive: checking whether a list is sorted is easy, but sorting a shuffled list requires a method. Computer programs are imperative knowledge written down. Declarative knowledge is used to check whether a program works correctly: that the output of a sorting program is really sorted is tested against the declarative definition.

Common mistake

Thinking that every sentence containing a formula or a definition is imperative knowledge. The sentence "$\sqrt{49} = 7$, because $7 \cdot 7 = 49$" states a result and its justification; it is declarative. The sentence has no steps, repetition, or stopping condition. A piece of knowledge is imperative only if it states the steps to follow to reach a result.

1.5.5.2 Algorithms

Not every kind of imperative knowledge can be given to a machine. "Make a nice meal" is an instruction but does not say which steps to follow. "Add salt until it tastes right" is a step, but how much salt to add is not specified. The instructions a machine can follow must be clear, executable, and finite.

Definition 1.5.7

An **algorithm** is a method that solves a problem in a finite number of clear and executable steps. An algorithm has these properties:

- **Clarity:** each step has a single meaning; two people carrying out the step do the same thing.
- **Executability:** each step can really be done with the tools at hand.
- **Finiteness:** the algorithm stops after a finite number of steps for every input.
- **Input and output:** the algorithm takes zero or more inputs and produces at least one output.

Algorithms turn imperative knowledge into clear, executable, finite steps. There is no single way to write an algorithm. English sentences, flowcharts, or **pseudocode**, which resembles a programming language but is not as strict, can be used. In these notes, as in the slides, algorithms are given in short pseudocode. In pseudocode, `while` means "repeat the lines below it as long as the condition is true," `return` means "give back the result and finish," and `print` means "write the result on the screen." Indented lines belong to the `while` line above them.

Common mistake

Forgetting the stopping condition. The instruction "add 1 to the number and repeat" is clear and executable but never ends. Such a sequence of instructions is not an algorithm; in a program it appears as an **infinite loop**. Every repeated step must have a stopping condition that will certainly be met some day.

1.5.5.3 Approximating a square root

An old and very effective method for finding square roots is attributed to Heron of Alexandria and is also known as the Babylonian method. The idea is this: if a guess is greater than the square root, then x divided by the guess is less than the square root; if the guess is less than the square root, the quotient is greater. The true square root lies between these two numbers. The average of the two is therefore usually a better guess than the guess itself.

Let us see it with an example. Let x = 25 and the guess 13. *25 ÷ 13 ≈ 1.92*. The true square root, 5, lies between 13 and 1.92. The average of the two numbers is *(13 + 1.92) ÷ 2 ≈ 7.46*; 7.46 is much closer to 5 than 13 is. The same operation is repeated with the new guess.

The pseudocode in the slides writes the method in five lines:

```
x = 25
guess = 1
while |guess * guess - x| > 0.01
    guess = (guess + x / guess) / 2
print(guess)
```

The lines do the following, in order:

1. `x = 25` : the number whose square root is sought is put into the variable x.
2. `guess = 1` : a rough starting guess is chosen. Any positive starting value works; 1 is the simplest choice.
3. `while |guess * guess - x| > 0.01` : the difference between the square of the guess and x, that is, the **error**, is computed. The vertical bars denote the **absolute value**: the sign of the difference is dropped and only its magnitude remains. If the error is greater than 0.01, the line inside the loop runs; if not, the loop ends. The value 0.01 is the "close enough" limit; such limits are called a **tolerance**.
4. `guess = (guess + x / guess) / 2` : the average of guess and x/guess becomes the new guess. Then we return to line 3 and check the error again.
5. `print(guess)` : when the loop ends, the final guess is written on the screen.

Each run of the loop is called an **iteration**. The slides trace the algorithm iteration by iteration for x = 25. Figure 1.5.8 shows the journey of the guess on the number line, and Table 1.5.10 shows the values in each iteration.



Faint dots: earlier guesses. Each iteration brings the guess closer to √x.

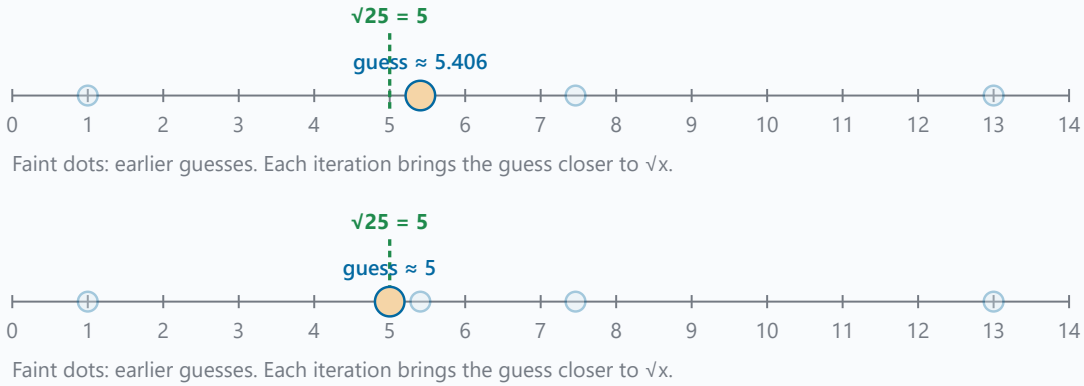


Figure 1.5.8: The position of the guess on the number line for x = 25; the dashed green line marks the true square root (5). (a) After iteration 1 the guess is 13; it has jumped from 1 to the other side of 5. (b) After iteration 3 the guess is ≈ 5.406. (c) After iteration 5 the guess is very close to 5; the faint dots are earlier guesses.

Iteration	Old guess	x / guess	New guess	Error $\lvert guess^2 - 25 \rvert$	Decision
start	—	—	1	24	continue
1	1	25	13	144	continue
2	13	1,9231	7,4615	30,6746	continue
3	7,4615	3,3505	5,4060	4,2251	continue
4	5,4060	4,6245	5,0152	0,1527	continue
5	5,0152	4,9848	5,0000	0,0002	halt

Table 1.5.10: Trace table for x = 25, starting guess 1. Values are rounded to four decimal places.

Three things in the table stand out. First, in the first iteration the error rose from 24 to 144. The guess jumped from 1 to 13 and passed 5. This is not a mistake. 1 was a very small guess and x/guess = 25 was very large; their average fell on the other side of the square root. After the second iteration the guess always stays above 5 and gets closer to it with every iteration.

Second, the speed of approach keeps increasing. The error fell from 144 to 30.67, then to 4.23, to 0.15, and to 0.0002. In the last iterations, each iteration roughly doubles the number of correct digits. That is why the method reached four-decimal accuracy in only five iterations. The final guess is not exactly 5 but about 5.00002; its error, however, is far below the tolerance.

Third, the stopping condition looks not at the guess itself but at its square. The algorithm does not know the true value of the square root; if it did, there would be no need to compute. The only thing it knows is the declarative definition: the square of the square root equals x. The stopping condition uses this definition. The imperative method moves forward, and the declarative definition decides when to stop.

Example 1.5.10

The same algorithm is run with x = 2 and guess = 1. How many iterations does it take, and what value is printed? How close is the printed value to the true $\sqrt{2} \approx 1.41421$?

Iteration	Old guess	x / guess	New guess	guess²	Error	Decision
start	—	—	1	1	1	continue
1	1	2	1.5	2.25	0.25	continue
2	1.5	1,3333	1,4167	2,0069	0,0069	halt

The algorithm stops after two iterations and prints about 1.4167. After the second iteration the error is 0.0069; since this is less than 0.01, the loop ends. The printed value is about *1.41667 − 1.41421 ≈ 0.0025* away from the true $\sqrt{2}$. If a more accurate result is wanted, the tolerance is reduced; for example, if 0.000001 is chosen, the algorithm makes two more iterations and finds 1.414214.

Common mistake

Choosing 0 as the starting guess. When the new guess is computed, x is divided by the guess; if the guess is 0, a division-by-zero error occurs and the algorithm cannot run. A negative x is also a problem: since no real number has a negative square, the error never drops below 0.01 and the loop runs forever. A program that uses the algorithm must check beforehand that the input is positive.

1.5.5.4 Euclid's algorithm

One of the oldest known algorithms appears in *Elements*, written by the Greek mathematician Euclid around 300 BC. The algorithm finds the greatest common divisor of two positive integers. Although it is about 2,300 years old, it is still used today, for example in reducing fractions and in the encryption methods that secure the internet.

Definition 1.5.8

The **greatest common divisor** (**GCD**) of two integers is the largest positive integer that divides both of them with no remainder. In the pseudocode of the slides this operation is written `GCD(a, b)`. The remainder of dividing a by b is written `a mod b`; the **mod** operation gives the remainder of a division.

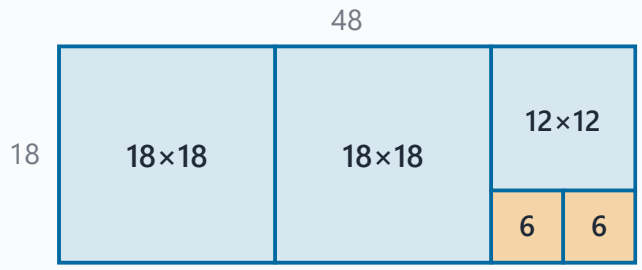
A method that applies the definition directly is conceivable: start from the smaller number, try every number downward, and stop at the first number that divides both. For 48 and 18 the trials go 18, 17, 16, … and stop at 6; 13 trials are needed. This method is correct but slow. For 1,000,000 and 999,999 it makes 999,999 trials, because the GCD of these two numbers is 1. Euclid's method answers the same question in only two iterations.

The pseudocode of Euclid's algorithm is:

```
GCD(a, b)
while b ≠ 0
    r = a mod b
    a = b
    b = r
return a
```

The algorithm works with two variables: a and b. In each iteration of the loop, the remainder of dividing a by b is first computed and put into r. Then the old b takes the place of a, and the remainder r takes the place of b. In this way each iteration turns the problem into the same problem with smaller numbers. When b becomes zero, the loop ends and the value in a is returned.

Let us look briefly at the mod operation. *48 mod 18* is the remainder of dividing 48 by 18. 18 fits into 48 twice: *2 × 18 = 36*. What is left is *48 − 36 = 12*. So *48 mod 18 = 12*. In general one writes $a = q \times b + r$, where q is the quotient, r is the remainder, and $0 \le r < b$.



48 = 2×18 + 12 ; 18 = 1×12 + 6 ; 12 = 2×6

The square that tiles everything exactly: 6×6

Figure 1.5.9: The geometric interpretation of Euclid's algorithm. From a 48 × 18 rectangle, two 18 × 18 squares are cut first, leaving a 12 × 18 strip. From it a 12 × 12 square is cut, leaving a 12 × 6 strip. That is filled exactly by two 6 × 6 squares. The largest square that tiles the rectangle exactly is 6 × 6.

Figure 1.5.9 gives a visible interpretation of the algorithm. We want to tile a rectangle 48 units long and 18 units wide exactly with equal squares. The side of the largest square that can be used is the GCD of 48 and 18. As squares with side 18 are cut from the rectangle, a smaller rectangle remains. Each cut corresponds to one iteration of the algorithm. At some point the remaining rectangle is filled exactly with congruent squares; the side of those squares is the number sought.

1.5.5.5 Euclid's algorithm step by step

Let us trace the call GCD(48, 18) line by line. At the start, a = 48 and b = 18.

1. **Iteration 1.** The condition b ≠ 0 is true (18 ≠ 0). r = 48 mod 18 = 12: two pieces of length 18 are taken out of 48, leaving 12. Then a = 18 and b = 12. The smaller number has now become the dividend.
2. **Iteration 2.** 12 ≠ 0 is true. r = 18 mod 12 = 6: one piece of length 12 is taken out of 18, leaving 6. Then a = 12 and b = 6.
3. **Iteration 3.** 6 ≠ 0 is true. r = 12 mod 6 = 0: two pieces of length 6 are taken out of 12, leaving nothing. Then a = 6 and b = 0.
4. **End of the loop.** The condition 0 ≠ 0 is false; the loop ends. The line `return a` returns 6.

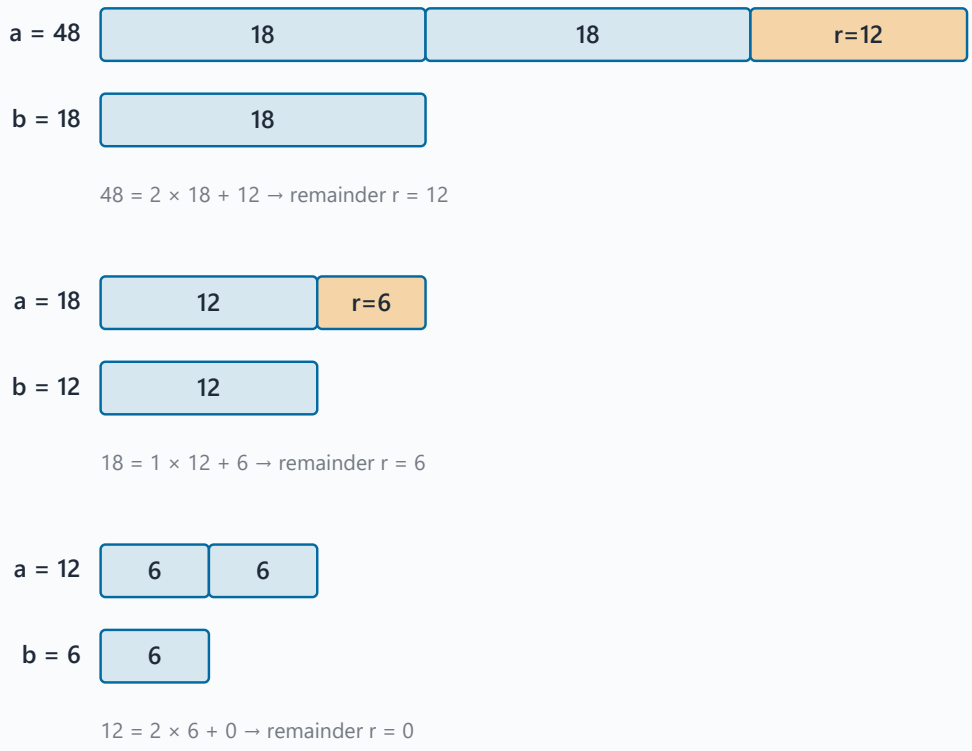


Figure 1.5.10: The remainder computation in each iteration for GCD(48, 18); the lengths of the bars are the magnitudes of the numbers. (a) Iteration 1: 48 = 2 × 18 + 12. (b) Iteration 2: 18 = 1 × 12 + 6. (c) Iteration 3: 12 = 2 × 6 + 0; the remainder is zero.

Figure 1.5.10 shows how many pieces of length b come out of the bar a in each iteration and what is left over. The remainder takes the place of b in the next iteration. The whole run is summarized in Table 1.5.11.

Iteration	a	b	b ≠ 0?	Division	r = a mod b	New a	New b
1	48	18	true	48 = 2 × 18 + 12	12	18	12

Iteration	a	b	b ≠ 0?	Division	r = a mod b	New a	New b
2	18	12	true	18 = 1 × 12 + 6	6	12	6
3	12	6	true	12 = 2 × 6 + 0	0	6	0
—	6	0	false	The loop ends; return a → 6			

Table 1.5.11: Trace table for GCD(48, 18). Each row shows the values of a and b at the start of an iteration, the computed remainder, and the new values at the end of the iteration.

Result: GCD(48, 18) = 6. Checking is easy: *48 = 8 × 6* and *18 = 3 × 6*. 8 and 3 have no common divisor greater than 1; so there is no common divisor greater than 6 either. The algorithm finished in three iterations. The method that applies the definition directly had made 13 trials.

Example 1.5.11

If GCD(270, 192) is computed with Euclid's algorithm, how do a, b, and r change in each iteration? What is the result?

Iteration	a	b	Division	r
1	270	192	270 = 1 × 192 + 78	78
2	192	78	192 = 2 × 78 + 36	36
3	78	36	78 = 2 × 36 + 6	6
4	36	6	36 = 6 × 6 + 0	0

At the end of the fourth iteration, a = 6 and b = 0; the loop ends and 6 is returned. Check: *270 = 45 × 6*, *192 = 32 × 6*; 45 and 32 have no common divisor greater than 1. With this result, the fraction *192/270* is reduced by dividing numerator and denominator by 6, giving *32/45*.

Common mistake

Swapping the order of the two assignment lines. In the loop, `a = b` is written first, then `b = r`. If `b = r` is written first, the old value of b is lost and the following line `a = b` puts r into a as well. For 48 and 18, after the first iteration a and b are both 12. In the second iteration 12 mod 12 = 0 comes out; this time a and b are both 0 and the algorithm returns 0 instead of 6. When updating variables in sequence, always check which value is still needed.

1.5.5.6 Why does Euclid's algorithm give the right answer and stop?

That an algorithm gives the right result on a few examples does not show that it works on every input. Two questions are asked about Euclid's algorithm: why does it always find the right result, and why does it always stop?

Theorem 1.5.1 (Euclid)

Let a and b be positive integers and $r = a \bmod b$. The common divisors of a and b are the same as the common divisors of b and r. Therefore $GCD(a, b) = GCD(b, r)$.

The justification of the theorem is as follows. The division is written $a = q \times b + r$; from this we get $r = a - q \times b$. If a number d divides both a and b, it also divides the difference $a - q \times b$, that is, it divides r too. Conversely, if d divides both b and r, it divides the sum $q \times b + r$, that is, a as well. So the two pairs of numbers have the same common divisors. Two pairs with the same common divisors also have the same greatest common divisor.

In the example this can be seen with numbers. The common divisors of 48 and 18 are 1, 2, 3, and 6. The common divisors of 18 and 12 are also 1, 2, 3, and 6. The numbers got smaller but the common divisors did not change. Each iteration of the algorithm shrinks the problem while preserving the GCD. In the last iteration the second number becomes 0. Since every positive number divides 0, the common divisors of a and 0 are the divisors of a, and the largest of them is a itself. That is why the algorithm returns a.

The answer to the question of stopping comes from a property of the remainder. The remainder is always smaller than the divisor: $0 \le r < b$. Since r takes the place of b in each iteration, b strictly decreases in every iteration. b is a nonnegative integer. A nonnegative integer cannot decrease forever; after decreasing at most b times, it reaches 0. So the loop ends after a finite number of iterations on every input. In practice the number of iterations is far smaller than b. The remainders at least halve every two iterations. So even for numbers below a million, no more than about 30 iterations are needed; the worst-case pair 832,040 and 514,229 finishes in 28 iterations.

Example 1.5.12

What happens if the algorithm is called with GCD(36, 84), that is, with the smaller number given first?

1. **Iteration 1.** 36 mod 84 = 36; 84 does not fit into 36 at all, and the whole number remains. Then a = 84 and b = 36. The numbers have swapped places.
2. **Iteration 2.** 84 mod 36 = 12 ($84 = 2 \times 36 + 12$). a = 36, b = 12.
3. **Iteration 3.** 36 mod 12 = 0. a = 12, b = 0 and the loop ends.

The algorithm returns 12. The order of the numbers does not change the result; only one extra iteration is spent swapping the numbers. The algorithm therefore does not need to require the inputs to be given in order.

Common mistake

Thinking that calling the algorithm with b = 0 gives an error. In the call GCD(15, 0) the loop condition is false at the first check; the loop never runs and 15 is returned. This is correct, because 15 divides both 15 and 0. The real problem is the call GCD(0, 0): since every number divides 0, the greatest common divisor is undefined, but the algorithm returns 0. That is why the definition of the algorithm says “two positive integers.”

1.5.6 Computational thinking

The square-root and Euclid algorithms were ready-made solutions to ready-made problems. Real-life problems, however, often arrive in a muddle: “We need a system that makes it easier for students to register for courses.” The path from such a request to a working program goes through several tasks. The problem is split into parts, the similarities between the parts are seen, what matters is separated from what does not, and finally the steps are written. This way of thinking is called computational thinking. This part takes the four components of computational thinking one by one and in the end combines them in an example.

1.5.6.1 What is computational thinking?

Definition 1.5.9

Computational thinking is the approach of turning problems into structures that a computer or a person can solve systematically. It has four components: **decomposition**, **pattern recognition**, **abstraction**, and **algorithm design**.

The concept was popularized by a short paper that the computer scientist Jeannette Wing published in 2006. According to Wing, computational thinking is a fundamental skill that everyone, not only programmers, should learn, just like reading, writing, and arithmetic. Thinking computationally does not mean thinking like a computer. A computer only does what it is told; computational thinking is the skill of working out what to tell it.

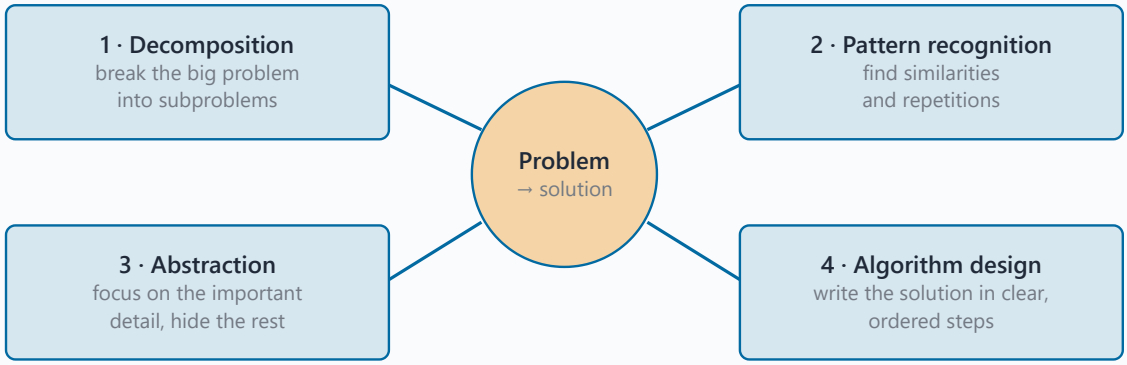


Figure 1.5.11: The four components of computational thinking around the problem: (1) decomposition breaks the big problem into subproblems; (2) pattern recognition finds similarities and repetitions; (3) abstraction focuses on the important detail and hides the rest; (4) algorithm design writes the solution in clear and ordered steps.

Figure 1.5.11 numbers the components in an order, but in practice this order is not rigid. While an algorithm is being written, a new subproblem may be noticed and one returns to decomposition. A pattern may determine what abstraction focuses on. The four components work together and feed one another.

Traces of computational thinking can also be seen in daily life. A cook divides a big meal into preparation, cooking, and serving stages; that is decomposition. A teacher uses the same extra worksheet for students who make similar mistakes every year; that is pattern recognition. A city map shows the streets, not the colors of the buildings; that is abstraction. A recipe is also an algorithm.

1.5.6.2 Decomposition

Definition 1.5.10

Decomposition is breaking a large, complex problem into smaller subproblems that are easier to solve. The input and output of each subproblem are defined separately.

The example in the slides is an online registration system. Users will register from a web page, their information will be stored, and a confirmation email will be sent to them. It is hard to think of this system as a single piece: it is not clear where to start, and when an error occurs it cannot be found where it is.

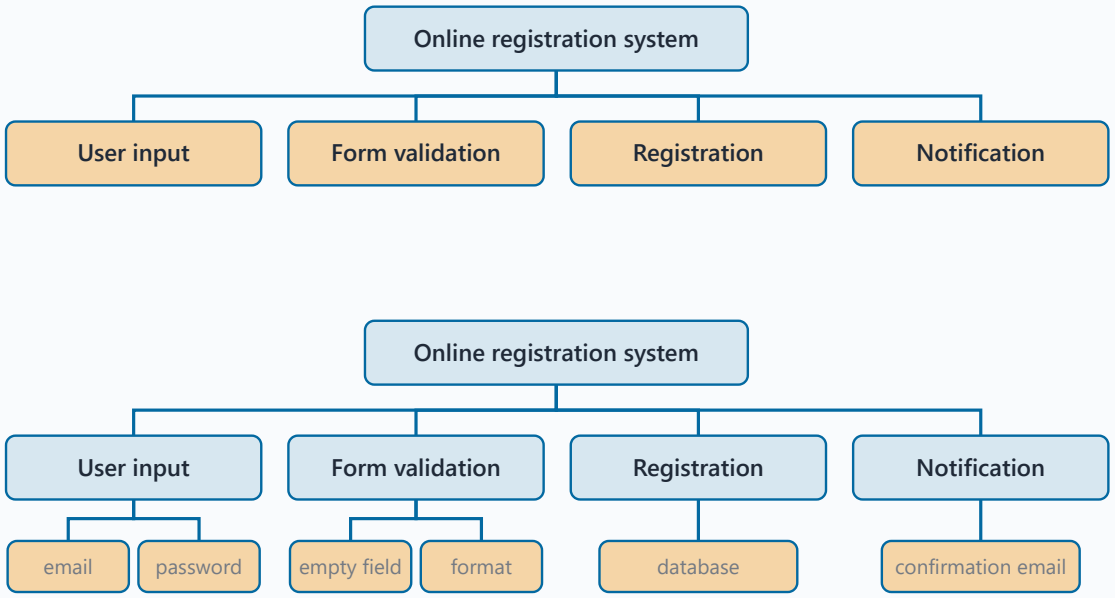


Figure 1.5.12: Decomposition of the online registration system. (a) First level: the system is divided into the subtasks user input, form validation, registration, and notification. (b) Second level: each subtask is divided into smaller jobs if needed; for example, form validation consists of an empty-field check and a format check.

Figure 1.5.12 shows two levels of decomposition. At the first level the system is divided into four main subtasks. At the second level some subtasks are divided again. Each leaf at the bottom of the tree is a small job that can be written and tested on its own. For example, the “format check” might be a small piece that only looks at whether the email address contains an @ sign and whether the password is at least eight characters.

Subtask	Input	Output
User input	Email and password	Signed in, or an error message
Form validation	The fields in the form	“Valid,” or a list of the faulty fields
Registration	Validated form data	A new record in the database and a registration number
Notification	Email address and registration number	Confirmation email sent to the user

Table 1.5.12: The subtasks of the registration system and the input and output of each.

Table 1.5.12 shows the real value of decomposition: the subtasks are linked to one another through their inputs and outputs. The output of form validation is the input of registration. The registration number that registration produces is the input of notification. When these links are written down explicitly, three important benefits appear.

- **Separate development and testing.** Each subproblem can be developed and tested separately. The person writing the notification part does not have to know how form validation works; they only need to know what information will come to them.
- **Clear interfaces.** The inputs and outputs of the subproblems are defined explicitly. When one part is changed, the other parts are not affected as long as its input and output stay the same.

- **Early view of dependencies.** It is noticed early which part waits for which. Notification cannot run before registration is finished, because it needs the registration number. This information determines the order in which the jobs are done.

Example 1.5.13

A department wants a program that prepares a seating plan for a final exam. There are 250 students; the halls have capacities of 80, 80, 60, and 40. How is the problem decomposed, and how are the students distributed among the halls?

Subtask	Input	Output
Read the student list	File taken from the student information system	List of 250 students
Read the halls	Hall names and capacities	List of 4 halls, total capacity
Check the capacity	Number of students, total capacity	"Enough" or "not enough" warning
Assign the students	Student list, hall list	A hall and seat number for each student
Print the lists	Assignment result	The list to be posted at the door of each hall

1. **Capacity check.** $80 + 80 + 60 + 40 = 260 \geq 250$; the capacity is sufficient and 10 seats remain empty.
2. **Assignment.** If the halls are filled in order, 80 students go to the first hall, 80 to the second, and 60 to the third. These three halls hold $80 + 80 + 60 = 220$ students. The remaining $250 - 220 = 30$ students go to the fourth hall; there $40 - 30 = 10$ seats remain empty.

Decomposition also revealed a dependency: the assignment cannot be done before the capacity check. If there were 270 students, the capacity check would give a "not enough" warning and the assignment would never start.

Common mistake

Listing the subtasks only by name, without writing their inputs and outputs. A list that says "validation," "registration," "notification" does not say what the parts give one another. When the parts are written separately and put together, one may expect a phone number while another sends an email address. For each subtask the question "what does it take, what does it give?" must be answered.

1.5.6.3 Pattern recognition

Definition 1.5.11

Pattern recognition is finding the similarities and repetitions among different problems or among different parts of one problem.

Let us return to the registration system. The system will register three kinds of people: students, staff, and visitors. A separate registration form could be designed for each. But when the forms are put side by side, something catches the eye.

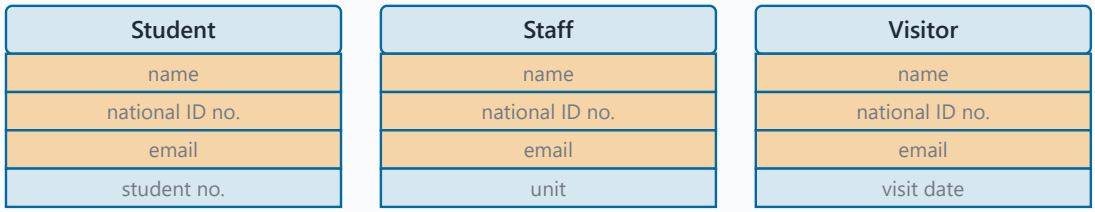


Figure 1.5.13: Three registration types: student, staff, and visitor. The name, national ID number, and email fields repeat in all three; the highlighted rows are these common fields. Each type has only one field of its own: student number, unit, and visit date.

Figure 1.5.13 shows the repeating structure. Three of the four fields of the three forms are the same. This is a pattern. Noticing the pattern has practical value. The code that checks that the name field is not left empty, that the ID number has 11 digits, and that the email address is written in the correct format does not have to be written three times. It is written once and used in all three forms. When an error is found in the ID number check, it is fixed in a single place.

Patterns are seen not only in data but also in operations. The average computation and Euclid's algorithm do different jobs, but both follow the same pattern: repeat an operation as long as a condition is true, and give the result when the condition becomes false. The square-root algorithm fits this pattern too. This pattern is so common that every programming language has a special structure for it, the loop.

Example 1.5.14

A dormitory administration wants a program that computes the monthly water, electricity, and natural gas bills. Let this month's values be as follows (the prices are for illustration): water 12 m^3, unit price 30 TL, fixed fee 40 TL; electricity 180 kWh, unit price 3 TL, fixed fee 60 TL; natural gas 90 m^3, unit price 8 TL, fixed fee 20 TL. What pattern is there in the three computations?

1. **Water.** *12 × 30 + 40 = 360 + 40 = 400* TL.
2. **Electricity.** *180 × 3 + 60 = 540 + 60 = 600* TL.
3. **Natural gas.** *90 × 8 + 20 = 720 + 20 = 740* TL.
4. **Pattern.** All three computations fit the same template: *amount = consumption × unit price + fixed fee*. Only three numbers change.

The total bill is *400 + 600 + 740 = 1,740* TL. Once the pattern is noticed, the program contains a single computation rule instead of three separate computations, and this rule is run three times with different numbers. If the dormitory later wants to add the internet bill, it is enough to give the same rule three new numbers.

Common mistake

Mistaking a superficial similarity for a pattern. A student number and an ID number both consist of digits, but one is an 11-digit national ID number with its own rules and the other is a number specific to the university. Putting them through the same check gives wrong results. A pattern means similarity not only in appearance but also in meaning and rules.

1.5.6.4 Abstraction

Definition 1.5.12

Abstraction is focusing on the details that matter for a purpose and hiding unnecessary details. The product of abstraction is a simplified **model** of reality.

Pattern recognition finds the repeating structure; abstraction gives this structure a name and separates it from the details. In the registration example, the common fields of the three types are gathered into a single "Person" model.

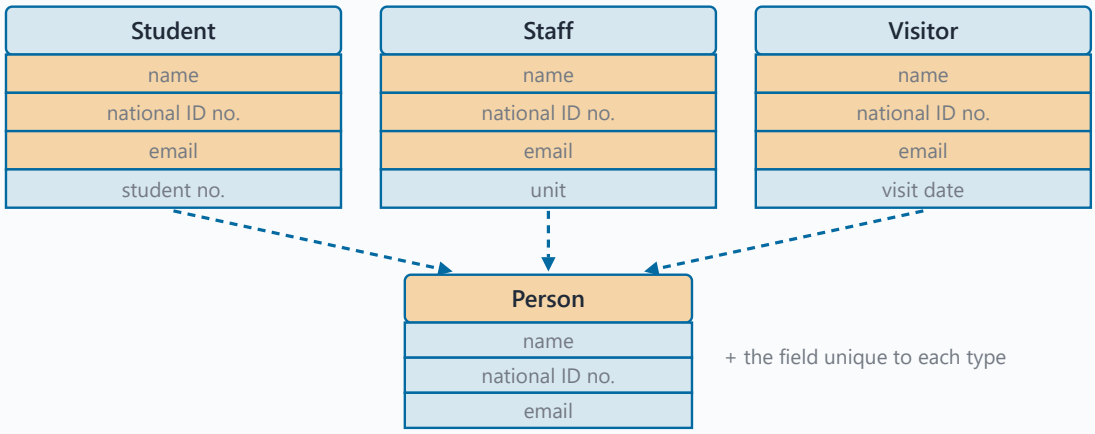


Figure 1.5.14: Abstraction. The common fields of the three registration types (name, national ID no., email) are gathered into a single Person model. Each type adds only its own field to the Person model.

Figure 1.5.14 shows the result. The Person model has three fields. A student record has all of Person's fields; in addition it has a student number. Staff add a unit, and visitors add a visit date. Most of the system, for example the notification part that sends email, works only with the Person model. The notification part does not have to know whether the person it emails is a student or a staff member; it needs only the email field. The differences show only where they are needed, for example in the part that prepares the student list.

A good abstraction factors out the repeating structure without losing the important differences. Putting the student number into the Person model as well would be wrong, because a visitor has no student number and this field would stay empty in most records. Removing the email field from the model would also be wrong, because the notification part could not work. The measure of abstraction is the purpose: the detail that serves the purpose stays, and the detail that does not is hidden.

Example 1.5.15

What features of the building does the fire escape plan posted in the corridors of a faculty building show, and which does it hide? Why does this abstraction work?

1. **Purpose.** In a fire, the plan must let someone in the building get outside by the shortest route.
2. **Details kept.** Corridors, stairs, exit doors, the locations of fire extinguishers, and the "You are here" marker.
3. **Details hidden.** The furniture in the rooms, the color of the walls, the number of windows, whom the rooms belong to, the year the building was built.
4. **Why does it work?** In a moment of panic, the reader's attention goes only to the way out. If the furniture were on the plan, the drawing would be crowded and the way out would be found later.

The plan an architect uses for the same building is far more detailed; it also shows wall thicknesses and electrical lines. There is no single "right" model of a building. Every purpose calls for its own abstraction.

Common mistake

Thinking that abstraction means "deleting details." In abstraction, details do not disappear; they are hidden. The student number is not visible in the Person model but stays in the student record. When needed, for example when a student's number is looked up, that detail is reached again. A model that really deletes a needed detail becomes useless.

1.5.6.5 Algorithm design and testing

Definition 1.5.13

Algorithm design is writing the solution of a problem as clear, ordered steps. **Testing** is checking whether the written steps produce the expected output on different inputs. Each pair of an input and an expected output chosen for checking is called a **test case**.

Algorithm design is not a job that is finished in one go; it proceeds in a cycle.

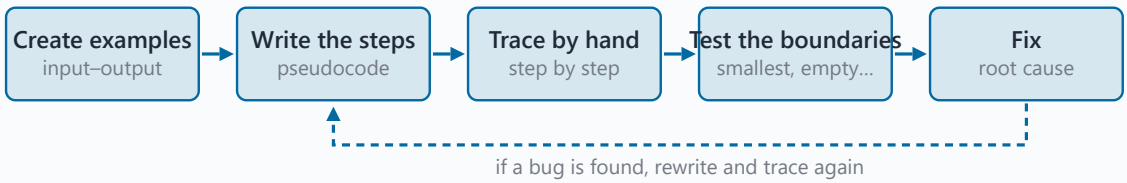


Figure 1.5.15: The algorithm design and testing cycle: examples made up of inputs and expected outputs are prepared, the steps are written in pseudocode, traced by hand step by step, boundary cases are tried, and the root cause of any bug found is fixed. If a bug was found, the steps are rewritten and traced again.

The cycle in Figure 1.5.15 has five stages.

1. **Creating examples.** Before writing code, a few inputs and the expected outputs for them are prepared. For the average: the expected output for the input 70, 85, 60, 75 is 72.5. The expected output is computed by hand; the algorithm's result will be compared with it.
2. **Writing the steps.** The solution is written in pseudocode.
3. **Tracing by hand.** The pseudocode is run on paper, line by line, with the prepared sample input. At each step the values of the variables are written into a **trace table**. The tables used in these notes for the Turing machine, the square root, and Euclid's algorithm are tables of this kind.

4. **Testing the boundaries.** Besides normal inputs, the smallest, the largest, and invalid inputs are tried. Inputs of this kind are called **edge cases**. Most bugs show up on these inputs.
5. **Fixing.** When a bug is found, it is not only the symptom but the **root cause** of the bug that is fixed. Then the cycle starts again: the corrected algorithm is traced again with all the examples.

Let us follow this cycle with an example. Suppose the following algorithm, which finds the average of a list of grades, has been written:

```
total = 0
count = 0
while there is a grade to read
    total = total + grade
    count = count + 1
print(total / count)
```

The algorithm does exactly the same computation as in Example 1.5.1: the total and the counter start at 0, each grade is added to the total and the counter goes up by one; at the end the total is divided by the counter. Now let us test it with inputs of different kinds.

Kind	Input	Expected output	Algorithm's output	Result
Normal	70, 85, 60, 75	72.5	290 / 4 = 72.5	passed
Smallest	100 (a single grade)	100	100 / 1 = 100	passed
Extreme values	0, 100	50	100 / 2 = 50	passed
Invalid	(no grades)	"No grades entered" warning	0 / 0: division by zero error	failed

Table 1.5.13: Test cases for the averaging algorithm.

Table 1.5.13 caught a bug in the last row. If no grade is entered, the loop never runs; the total and the counter stay at 0 and the last line divides 0 by 0. The program crashes. Tests with normal inputs would never have revealed this bug.

There are two ways to fix the bug. The first is to remove the symptom: "if the counter is 0, take the average to be 0." The program no longer crashes but says something wrong: the average of a student who has no grades at all appears as 0 and the student is taken to have failed the course. The second is to fix the root cause. The root cause is this: the average of an empty list is undefined, and the algorithm never considered this case. The right fix is to handle this case separately:

```
total = 0
count = 0
while there is a grade to read
    total = total + grade
    count = count + 1
if count = 0
    print("No grades entered")
else
    print(total / count)
```

In the corrected algorithm, the `if` line makes a decision: if the counter is 0, the warning is printed; otherwise (`else`) the average is computed. When the algorithm is traced again with the four test cases, the first three again give 72.5, 100, and 50; the fourth now prints the expected warning.

Common mistake

Testing only with normal inputs. Hundreds of randomly chosen ordinary lists would not reveal the empty-list bug even once. What matters is not the number of test cases but their variety. A good test set contains at least one normal, one smallest, one extreme, and one invalid input.

1.5.6.6 All four together: a grade calculator

How do the four components of computational thinking work together in solving a problem from start to finish? Let us see this with a small problem. An instructor wants a simple tool that computes students' end-of-term scores. The score is 40% of the midterm grade plus 60% of the final exam grade. A student whose score is 50 or above passes.

Decomposition. The problem is split into four subtasks. Reading the grades: the input is the student's midterm and final grades, the output is two numbers. Validating the grades: the input is two numbers, the output is "valid" or "invalid." Computing the score: the input is two valid grades, the output is the score. Decision: the input is the score, the output is "passed" or "failed."

Pattern recognition. The midterm and final grades follow the same rules: both must be between 0 and 100. The validation rule is written once and applied to both grades. The score computation is also a pattern: each grade is multiplied by its own weight and the results are added. If an assignment grade is added later, one more term is added to the same template.

Abstraction. For the tool, a student consists only of two numbers: the midterm and the final. The student's name, department, attendance record, or the answers on the exam paper do not affect this computation and have no place in the model.

Algorithm design. The steps are written in pseudocode:

```
read(midterm, final)
if midterm < 0 or midterm > 100 or final < 0 or final > 100
    print("Invalid grade")
else
    score = (40 * midterm + 60 * final) / 100
    if score >= 50
        print(score, "passed")
    else
        print(score, "failed")
```

In the score formula, the percentages are written as 40 and 60 instead of 0.4 and 0.6, and the result is divided by 100. The two forms are the same in mathematics. However, computers cannot store decimal numbers like 0.4 exactly in binary, and small rounding differences can occur. For example, the computation *55 × 0.4 + 72 × 0.6* may give 65.19999… instead of 65.2 on a computer. Multiplying with whole numbers and dividing at the very end reduces this risk.

Example 1.5.16

The grade calculator is tested with the following inputs. What does the tool print for each input?

Kind	Midterm	Final	Score	Output
Normal	55	72	(40 × 55 + 60 × 72) / 100 = (2.200 + 4.320) / 100	65.2 passed
Normal	40	45	(1.600 + 2.700) / 100	43 failed
Boundary	80	30	(3.200 + 1.800) / 100	50 passed
Extreme value	100	100	(4.000 + 6.000) / 100	100 passed
Extreme value	0	0	(0 + 0) / 100	0 failed
Invalid	−5	70	Midterm below 0	Invalid grade
Invalid	60	105	Final above 100	Invalid grade

The third row is the most instructive. The score is exactly 50 and the rule says “50 and above passes.” If `score > 50` had been written by mistake instead of `score >= 50` in the algorithm, this student would appear to have failed. Normal inputs never reveal this difference; only the input exactly on the boundary does.

In this small example each of the four components did a job. Decomposition divided the work into parts and showed which part waits for which. Pattern recognition prevented the validation rule from being written twice. Abstraction left out unneeded student information. Algorithm design and testing caught, before any code was written, a bug that could have caused a student on the boundary to fail unfairly.

1.5.7 Summary

- **Computation** is the processing of data according to rules; it is not limited to arithmetic. Searching, sorting, decision making, transformation, and communication are computation too. Every computation has an **input**, a **process**, and an **output**; the process uses the resources time, memory, energy, and communication.
- The building blocks of computation are mathematics, logic, data, algorithms, hardware, and a programming language. An algorithm is an idea; a program is the writing of that idea in a particular language.
- **Computer science** studies the representation and processing of information and the methods of solving problems. Its main areas of interest are abstraction, algorithms, systems, and people and society. The responsibilities of technology are **accessibility**, **privacy**, **fairness**, and **sustainability**.
- From Babbage and Lovelace to Turing, from Colossus and ENIAC to today's phones, the basic ideas have stayed the same; as hardware became faster and smaller, these ideas spread to wider fields.
- In a **fixed-program** machine the job is determined by the physical layout of the machine. In a **stored-program** machine the instructions are kept in memory like data; loading a new program gives the machine a new job. This idea made the general-purpose computer possible.
- The **Turing machine** is an abstract model of computation made up of a tape, a read/write head, a state, and transition rules. Each step is determined only by the (state, symbol read) pair. The machine that adds 1 turns 111 into 1000 in four steps.
- The **universal Turing machine** reads the rules of another machine given on its tape and imitates it. This is the theoretical foundation of the stored-program idea.
- The limits of computers come from time, memory, energy, computability, data quality, and the model. Computability limits such as the **halting problem** cannot be overcome with any hardware.
- **Declarative knowledge** says what is true; **imperative knowledge** says how to reach a result. An **algorithm** turns imperative knowledge into clear, executable, finite steps.
- The square-root method proceeds by averaging the guess and x/guess and stops when the square of the guess gets close enough to x; for x = 25 it reaches 5.0000 in five iterations.
- **Euclid's algorithm** uses the equality *GCD(a, b) = GCD(b, a mod b)*; because the remainder shrinks in every iteration, it always stops. GCD(48, 18) gives 6 in three iterations.
- **Computational thinking** combines four components: **decomposition** (breaking into subproblems, defining inputs and outputs), **pattern recognition** (finding repetitions), **abstraction** (focusing on what matters and hiding the rest), and **algorithm design** (clear, ordered steps). The design is tested with normal, smallest, extreme, and invalid inputs; it is the root cause of a bug that is fixed, not its symptom.

1.5.8 Key terms

The key terms of this chapter are listed below.

computation input computer science accessibility privacy fairness sustainability fixed program stored program general purpose von Neumann architecture Turing machine tape symbol read/write head state transition rule universal Turing machine computability halting problem declarative knowledge imperative knowledge algorithm pseudocode iteration tolerance infinite loop Euclid's algorithm greatest common divisor (GCD) remainder modulo (mod) computational thinking decomposition pattern recognition abstraction algorithm design testing test case edge case trace table

1.5.9 Test Yourself

1. The Turing machine that adds 1 to a binary number is run with 1111 (binary 15) on the tape; the head is at the rightmost digit. In how many steps does the machine halt, which rules are used, and what is left on the tape?

   Answer

   Since all four digits are 1, rule 1 is applied four times; each time 0 is written and the head moves left. The tape becomes 0000 and the head arrives at the blank cell to the left of the number. In the fifth step □ is read; rule 3 writes 1 and the machine halts. 10000 is left on the tape: *16 = 15 + 1*. The number of steps is *4 + 1 = 5*; rule 2 is never used.

2. The parity machine is run with 10010 on the tape. What is the state after each symbol, and what does the machine write at the end?

   Answer

   The state starts as “even.” 1 is read: odd. 0: odd. 0: odd. 1: even. 0: even. In the sixth step the blank cell is read; since the state is “even,” E is written and the machine halts. The tape becomes 10010E. The string has two 1s, and two is even.

3. GCD(105, 75) is computed with Euclid’s algorithm. What are a, b, and r in each iteration? How

many iterations does it take, and what is the result?

Answer

Iteration 1: *105 = 1 × 75 + 30*, r = 30; a = 75, b = 30. Iteration 2: *75 = 2 × 30 + 15*, r = 15; a = 30, b = 15. Iteration 3: *30 = 2 × 15 + 0*, r = 0; a = 15, b = 0. The loop ends; 15 is returned after three iterations. Check: *105 = 7 × 15*, *75 = 5 × 15*; 7 and 5 have no common divisor greater than 1.

4. The square-root algorithm is run with x = 9 and guess = 1. What are the guess and the error after each iteration? After how many iterations does the algorithm stop?

Answer

At the start the error is *|1 − 9| = 8*. Iteration 1: *(1 + 9) ÷ 2 = 5*, error *|25 − 9| = 16*. Iteration 2: *(5 + 1.8) ÷ 2 = 3.4*, error *|11.56 − 9| = 2.56*. Iteration 3: *(3.4 + 2.6471) ÷ 2 ≈ 3.0235*, error ≈ 0.1417. Iteration 4: *(3.0235 + 2.9767) ÷ 2 ≈ 3.0001*, error ≈ 0.0005. Since the error drops below 0.01, the algorithm stops after four iterations and prints about 3.0001.

5. Is each of the following pieces of knowledge declarative or imperative? (a) Water boils at 100 °C at sea level. (b) To find out whether a number is divisible by 3, add its digits; if the sum is divisible by 3, so is the number. (c) The area of a rectangle is the product of its side lengths.

Answer

(a) is declarative; it states a fact. (b) is imperative; it gives the steps to follow to reach a result: add the digits, check the sum. (c) is declarative; it defines what the area is. This definition can be used to compute an area, but the sentence does not describe a method, it states an equality.

6. How do a mechanical music box that plays a single song with the pins on its cylinder and a smart speaker that plays songs downloaded from the internet differ in terms of program?

Answer

The music box is fixed-program: the song it plays is the physical arrangement of the pins on the cylinder; for another song the cylinder must be changed. The smart speaker is stored-program: the processor inside runs software stored in memory and can gain new features through software updates. The smart speaker itself is still designed for particular jobs and the user cannot load their own programs onto it; so it is not counted as a general-purpose computer.

7. Which limit of computers does each of the following situations illustrate? (a) A student tries to write a tool that tells in advance whether any program will enter an infinite loop. (b) A program that makes 10^9 attempts per second can try all the possibilities of a 50-bit key in about 13 days in

the worst case. (c) On a survey form, 250 was entered in the age field and the average age came out meaningless.

Answer

(a) is the computability limit; this is the halting problem and such a general tool cannot be written. (b) is the time limit: $2^{50} \div 10^{9} \approx 1.13 \times 10^{6}$ seconds, about 13 days; faster hardware shortens the time, but each bit added to the key doubles it again. (c) is the data-quality limit; even if the algorithm works correctly, faulty input gives a wrong result.

8. An algorithm that counts the words in a text is to be tested. Besides a normal sentence, which boundary and invalid cases should be tried? Give at least four cases and the expected result for each.

Answer

A sample test set: empty text → 0; a single word (“Hello”) → 1; more than one space between words (“one␣␣two”) → 2; spaces at the beginning and end of the text (“␣one two␣”) → 2; text consisting only of spaces → 0. If the algorithm increments the counter at every space, it gives a wrong result in the last three cases. The root cause is counting spaces instead of words; the right fix is to count the places where a word starts.

1.5.10 Further reading

- J. M. Wing, “Computational Thinking”, *Communications of the ACM*, 49(3), 2006: a short paper, in three pages, that explains why computational thinking is a basic skill for everyone and popularized the concept.
- J. V. Guttag, *Introduction to Computation and Programming Using Python* (3rd ed., MIT Press, 2021), Chapter 1: the distinction between declarative and imperative knowledge, the square-root method, and fixed-program and stored-program machines are explained in language very close to that of these notes.
- C. Petzold, *The Annotated Turing* (Wiley, 2008): explains Turing’s 1936 paper line by line and builds up the Turing machine, the universal machine, and computability without requiring mathematical background.
- M. Sipser, *Introduction to the Theory of Computation* (3rd ed., Cengage, 2012), Chapters 3–4: a precise mathematical treatment of Turing machines and the halting problem; for those who want to continue the subject at an advanced level.

1.5.11 References

1. J. M. Wing, “Computational Thinking”, *Communications of the ACM*, 2006.

2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, MIT Press.

3. A. B. Downey, C. Mayfield, *Think Java*, Green Tea Press.

4. M. Sipser, *Introduction to the Theory of Computation*, Cengage.

PART I · CHAPTER 2

Operating Systems

The operating system manages hardware resources and provides services to applications. The first deck starts with kinds of software and licensing models, then covers booting, the kernel and system calls, processes and threads, time sharing, memory management and virtual memory, file systems, device drivers and user permissions. The second deck is hands-on with Windows: the desktop, windows and file management, absolute and relative paths, settings, security and system maintenance.

NOTES IN THIS CHAPTER

2.1 Software and Operating Systems

2.2 The Windows Operating System

PART I · CHAPTER 2.1

Software and Operating Systems

Hardware is useful only when a program tells it what to do. The most important of these programs is the operating system, which starts running when the computer is switched on and manages every resource until it is shut down. These notes first explain what software is, how it is classified, and how it talks to the hardware layer by layer. They then turn to the tasks of the operating system, the boot process, and the kernel. Process, memory, device, and file management are covered with step-by-step worked examples. Every intermediate step is shown for round-robin scheduling, virtual address translation, page replacement, and file reads. The last sections compare user interfaces, privilege management, types of operating systems, software licenses, updates, and virtualization.

LEARNING OUTCOMES

Types of software
Distinguish system software, application software, programming tools, and firmware by their examples; explain the layers between software and hardware.

Operating system
Explain the role of the operating system between the user, applications, and hardware; describe the boot process, the kernel, and a system call step by step.

Resource management
Interpret process states and scheduling, memory allocation and protection, virtual memory, device drivers, and the file system; compute Gantt charts, address translations, and page replacement.

Choosing a system
Compare personal, mobile, server, embedded, and real-time operating systems against requirements; evaluate licensing, update, and virtualization options.

2.1.1 Software

Hardware does nothing by itself; the processor, memory, and disk need instructions that tell them, step by step, what to do. All of these instructions together are called software.

2.1.1.1 What is software?

In everyday speech, “software” usually brings to mind just a program. A software product, however, contains more than a program. A game ships with picture, sound, and map files alongside its code. An accounting program reads tax rates from a table file. Both have installation instructions and a user manual. If any of these parts is missing, the software either does not run or cannot be used correctly.

Definition 2.1.1

Software is the whole of the programs that define what a computer does, the data those programs use, and the related documentation. **Hardware** is the physical, tangible parts of the computer: the processor, memory, disk, keyboard, and screen.

The division of labor between hardware and software is clear. Hardware carries out operations physically: the processor adds, memory stores, the screen displays. Software decides which operation is performed, when, and how. The same hardware becomes a word processor one day and a game machine the next, depending on the software loaded onto it. This flexibility is the basic property that distinguishes a computer from single-purpose machines.

The creation of a piece of software also follows a chain: **user requirement → algorithm → program code → running software → result**. The requirement “calculate the semester average” becomes the algorithm “take the grades, multiply by their weights, add them up”; the algorithm is written in a programming language and, once compiled and run, produces the average. A mistake in one link, for example a misunderstood requirement, is carried into all the following links.

2.1.1.2 Quality attributes of software

When choosing between two programs that do the same job, the question “does it work?” is not enough. Two calculator apps may both give the right answer, yet one makes you wait ten seconds at every launch while the other moves the keys around in every release. In software engineering, how good a piece of software is gets judged by six basic attributes.

Attribute	Meaning	Example in a banking app
Functionality	Performing the expected tasks correctly	A transfer amount is deducted from the account in full
Usability	Easy to learn and effective to use	Sending money takes three taps
Performance	Using time and resources efficiently	The account statement opens in one second
Reliability	Behaving predictably when something fails	If the connection drops, a transaction is never left half-done: it either completes or is cancelled
Security	Protection against unauthorized access and modification	Even if someone else knows the password, they cannot sign in without the code sent to the phone

Attribute	Meaning	Example in a banking app
Maintainability	Fixes and improvements can be managed	A new tax rule is added in a few days

Table 2.1.1: Quality attributes of software, with examples from a banking app.

Table 2.1.1 shows these attributes in a familiar app. The attributes sometimes conflict: an extra verification step added for security reduces usability. Good design strikes the right balance for the intended use; in aircraft control software reliability comes first, in a photo app usability does.

Common mistake

Treating reliability and security as the same thing. Reliability means the software behaves consistently in the face of faults: data is not corrupted when the power goes out. Security means protection against someone with malicious intent: a person without authorization cannot reach the data. A very reliable program can be insecure, and the reverse is also possible.

2.1.1.3 Layers between software and hardware

When a user presses the “Print” button in a word processor, they do not think about what voltage will drive the printer’s inkjet head. The word processor does not know it either. These details are hidden in lower layers. A computer system is organized as stacked layers, and each layer talks only to the one directly below it.

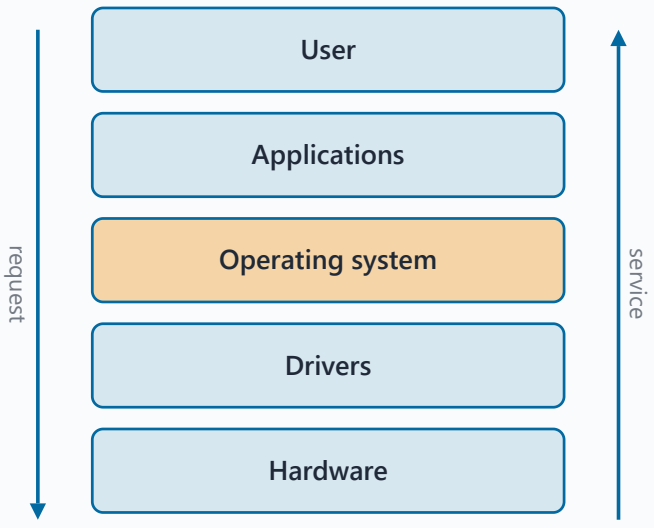


Figure 2.1.1: Five layers from the user to the hardware. Requests travel from top to bottom; results, that is, the services offered by the lower layer, travel from bottom to top.

Figure 2.1.1 shows the five layers. At the top is the **user**, who interacts only with applications. **Applications** ask the operating system for their work. The **operating system** passes these requests to the **drivers**. Drivers in turn produce the commands specific to each piece of hardware. At the bottom sits the **hardware**, which does the work physically.

Definition 2.1.2

Abstraction means a layer hides the details beneath it and offers the layer above a simpler, common service. The upper layer knows how to use the service; it does not have to know how the service is carried out.

The biggest gain from abstraction is portability. The word processor just says "print this page." Whether the printer is inkjet or laser, the request is the same. The driver layer absorbs the difference: each printer manufacturer writes a driver for its own device. Thanks to drivers, the same application runs on thousands of different hardware combinations. A programmer writing an application does not need to know every printer model.

2.1.1.4 Types of software

Software running on a computer falls into four main groups by task. The criterion is whom the software serves: the hardware and other programs, the user's work, the software developer, or a single device itself?

Definition 2.1.3

System software manages the hardware and provides a runtime environment for other software. **Application software** performs a specific task for the user. **Programming tools** support writing, translating, testing, and debugging code. **Firmware** is software embedded in a device's permanent memory that manages the device's basic functions.

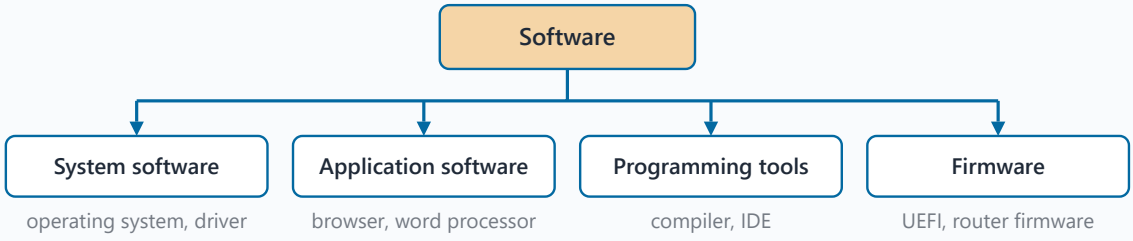


Figure 2.1.2: Types of software and their examples: system software (operating system, driver), application software (browser, word processor), programming tools (compiler, IDE), and firmware (UEFI, router firmware).

Figure 2.1.2 gathers the four types and their examples in a tree. The best-known example of **system software** is the operating system; device drivers and **utilities** such as disk cleanup and backup also belong to this group. **Application software** serves the user's actual goal: a browser opens web pages, a word processor writes reports.

Programming tools are used to build new software. A **compiler** translates the source code a person writes into machine code the processor can run. An **integrated development environment (IDE)** combines a code editor, a compiler, and a debugger in one window.

Firmware sits where hardware and software meet. The UEFI on a motherboard, a router's menu, and the code that handles a washing machine's program dial are all of this kind. Firmware is stored not on the disk but in the device's own permanent memory (ROM or flash memory). It starts running when the device is switched on, without anything being loaded. Users mostly do not notice it; they hear of it only when the manufacturer releases an update.

Example 2.1.1

Which type does each of the following software on a computer belong to? A printer driver, a Java compiler, a spreadsheet program, the software inside a smartwatch, a disk defragmentation tool.

1. **Printer driver:** manages hardware together with the operating system → system software.
2. **Java compiler:** translates source code so that new software can be produced → programming tool.
3. **Spreadsheet program:** does the user's calculation work → application software.
4. **Smartwatch software:** embedded in the device's permanent memory and manages its basic functions → firmware.
5. **Disk defragmentation tool:** deals not with a user's document but with the layout of the disk → system software (utility).

2.1.1.5 System and application software

Besides these four types, one more type comes up often: **middleware**. Middleware makes it easier for applications to communicate with the operating system and with each other. For example, a school's grading system reaches the database that holds its records through a connection layer; the programmer does not have to write out the details of the network connection.

Type	Examples	Main purpose
System	Operating system, driver, utility	Managing the computer and its resources
Application	Word processor, browser, game	Doing the user's task

Type	Examples	Main purpose
Middleware	Database connection layer, message queue	Making it easier for applications to communicate with the system and each other
Programming tools	Compiler, integrated development environment (IDE), debugger	Producing new software

Table 2.1.2: Examples and main purposes of the software types.

Table 2.1.2 summarizes the distinction. The boundaries are not always sharp. A browser behaves partly like a platform because it provides a runtime environment for web applications. Even so, the basic question is useful: is the software doing the user's work, or enabling other software to run? In the first case it is application software; in the second, system software.

2.1.2 The operating system

Dozens of programs run on a computer at the same time, and all of them want to use the same processor, memory, and disk. The software that brings order to this crowd is the operating system.

2.1.2.1 What is an operating system?

Definition 2.1.4

An operating system (OS) is the core system software that manages hardware resources and offers common services to applications. It is the first large piece of software loaded when the computer starts, and it keeps running until shutdown.

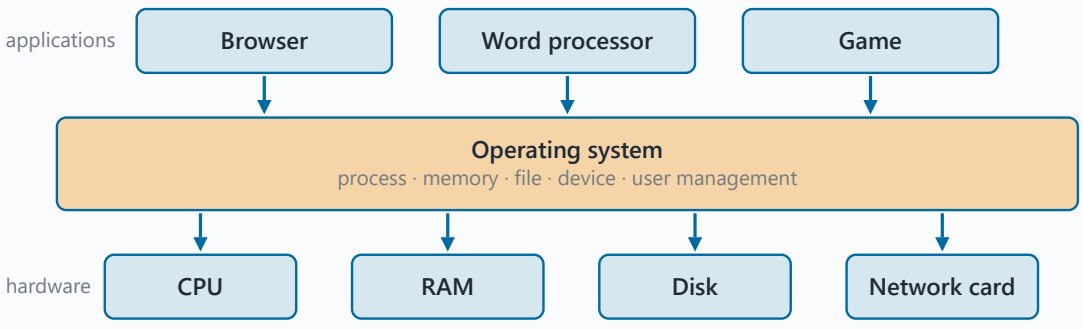


Figure 2.1.3: The operating system sits between applications (browser, word processor, game) and hardware (CPU, RAM, disk, network card). Process, memory, file, device, and user management are the job of this layer.

Figure 2.1.3 shows where the operating system sits. The applications above cannot reach the hardware without the operating system. The hardware below cannot choose by itself which application to serve. Between these two worlds the operating system does three basic jobs.

- **It abstracts hardware details.** An application says “open the file”; it does not need to know whether the file is on an SSD, a USB drive, or a server on the network.
- **It shares resources safely.** Many programs use the same processor and memory. The operating system gives each a turn and a region, and stops one program from writing into another’s memory.
- **It offers the user an interface.** A graphical interface with windows, or a command line driven by text commands, is the way the user talks to the computer.

2.1.2.2 Tasks of the operating system

The work of the operating system divides into six main tasks. Each task manages one kind of resource.

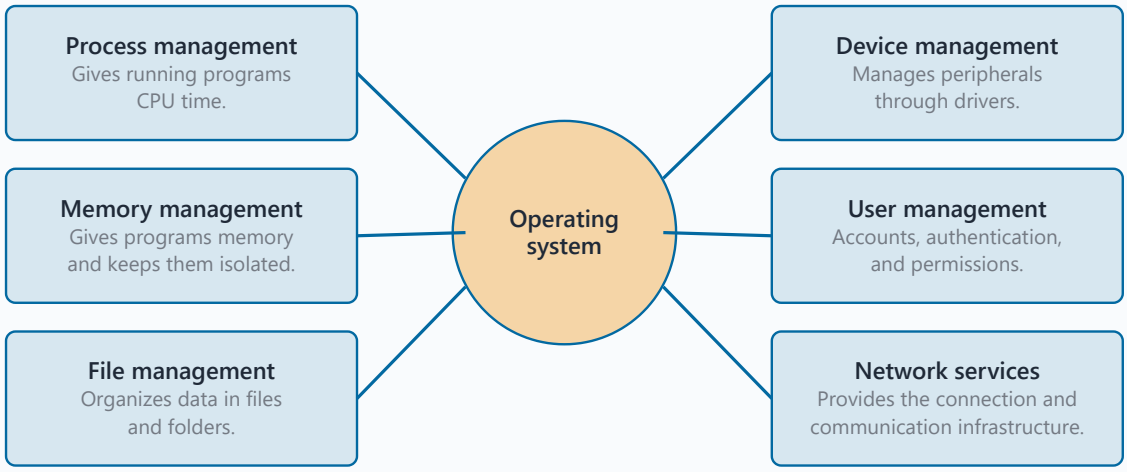


Figure 2.1.4: The six tasks of the operating system: process, memory, file, device, and user management, and network services.

Figure 2.1.4 shows these tasks together.

- **Process management:** gives running programs processor time. It decides which program uses the processor, when, and for how long.
- **Memory management:** gives programs memory and isolates them from one another. One program crashing does not corrupt another’s data.
- **File management:** organizes data in files and folders, and keeps track of where files sit on the disk and who may read them.
- **Device management:** manages peripherals such as the keyboard, printer, and graphics card through drivers.
- **User management:** manages accounts, authentication, and permissions.
- **Network services:** provides the infrastructure needed to establish connections and exchange data.

2.1.2.3 Booting: how does a computer start?

The operating system sits on the disk, but who loads it from the disk into memory? When the computer is off, RAM is empty. The moment the power button is pressed, there is no program in memory to run. The process that solves this "chicken-and-egg" problem is called booting.

Definition 2.1.5

Booting is the start-up process that runs from the moment power is applied until the operating system is ready for use, with each stage loading the next. The small program that moves the operating system from disk into RAM is called the **boot loader**.

The solution is a small memory that keeps its contents even when the power is off. The permanent memory on the motherboard (ROM or flash memory) holds firmware written by the manufacturer. When the processor receives power, it always starts running from a fixed address in this memory. Start-up proceeds in six stages.

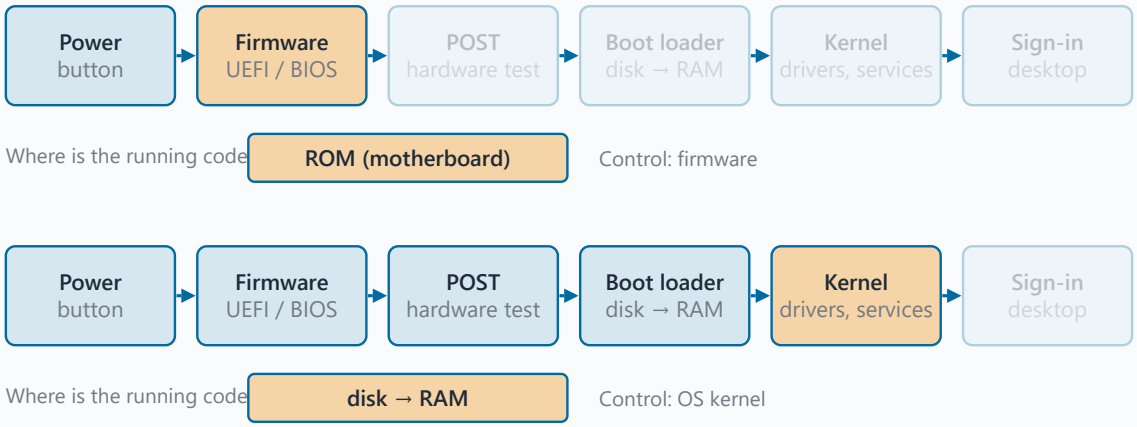


Figure 2.1.5: Boot stages. (a) At the start, control is with the firmware and the running code is in the ROM on the motherboard. (b) The boot loader has loaded the kernel from disk into RAM and handed control to it; the running code is now the operating system kernel in RAM.

Stage	What happens?	Where is the running code?	Who has control?
1. Power	When the voltages are stable, the power supply sends the motherboard a "power good" signal. RAM is empty.	—	User
2. Firmware	The processor starts running the UEFI (formerly called the BIOS) on the motherboard.	ROM (motherboard)	Firmware
3. POST	Power-on self-test: basic hardware such as memory, keyboard, and disk is checked.	ROM (motherboard)	Firmware

Stage	What happens?	Where is the running code?	Who has control?
4. Boot loader	The boot loader is loaded into RAM from the first suitable device in the boot order.	Disk → RAM	Boot loader
5. Kernel	The kernel is loaded into RAM; it starts the drivers, the first processes, and the system services.	Disk → RAM	OS kernel
6. Sign-in	Once identity is verified, the desktop and start-up applications open.	RAM	Operating system

Table 2.1.3: Where the running code is and who has control at each boot stage.

Table 2.1.3 follows the six stages with two questions: where is the code running at that moment, and who has control? Figure 2.1.5 shows two intermediate moments of this journey.

Power and firmware. The power button sends a run signal to the power supply. When the supply's voltages become stable, it sends a "power good" signal to the motherboard. On that signal the processor runs the firmware held in the permanent memory on the motherboard. In today's computers this firmware is **UEFI** (Unified Extensible Firmware Interface). In older computers the same job was done by the **BIOS** (Basic Input/Output System).

POST. The firmware's first job is the **power-on self-test (POST)**. It checks that the memory modules are installed and working and that the keyboard and disks are connected. If something is wrong, the computer emits a series of beeps or prints a message on screen. If no RAM module is installed at all, start-up stops here, because every later stage requires loading something into RAM.

Boot loader. The firmware settings contain a **boot order**, for example "USB first, then SSD." The firmware tries the devices in this order and skips any that are not attached. When it finds a suitable device, it looks at the small **EFI system partition** on the disk. The boot loader program sits in this partition, which is formatted with FAT32. On Windows the boot loader is called Windows Boot Manager (the file is *bootmgfw.efi*); on Linux the most common boot loader is GRUB. The boot loader is loaded into RAM and control passes to it.

Kernel. The boot loader finds the operating system kernel, loads it into RAM, and hands control to it. On Windows this takes two steps: the boot loader runs the program *winload.efi*, which in turn loads the kernel file *ntoskrnl.exe*. Once the kernel starts running, it loads the device drivers and starts the first processes and the system services that run in the background.

2.1.2.4 Secure Boot, TPM, and disk partitioning

Every link in the boot chain trusts the next one. The firmware runs the boot loader it finds on the disk, and the boot loader runs the kernel. If a malicious program can put itself in place of the boot loader, it runs before the operating system. From that point even antivirus scanners cannot see it, because they are loaded after it. Modern computers use two tools against this danger.

Definition 2.1.6

Secure Boot means that UEFI runs only a boot loader and kernel that carry a trusted digital signature. **TPM** (Trusted Platform Module) is a security chip that stores encryption keys and boot measurements. A TPM can be a separate chip on the motherboard, a unit inside the processor, or a component within the firmware.

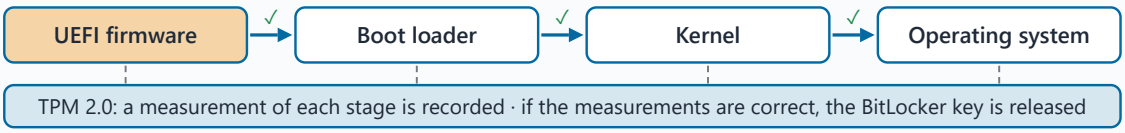


Figure 2.1.6: The chain of trust. The UEFI firmware checks the boot loader's digital signature, and the boot loader checks the kernel's (✓). The TPM records a measurement of each stage; if the measurements are correct, the disk-encryption (BitLocker) key is released.

Figure 2.1.6 shows the chain of trust from left to right. The first link in the chain is the firmware, which the manufacturer writes at the factory together with trusted keys. Before running the next link, each link checks its signature. A digital signature is a mathematical stamp showing that a file was produced by a particular organization and has not been modified since. If even a single byte of the file changes, the signature no longer matches. Because the signature of a modified boot loader cannot be verified, UEFI does not run it and start-up halts.

At every stage the TPM computes and records a short fingerprint of the loaded code. The disk-encryption tool BitLocker keeps its decryption key in the TPM; the TPM releases the key only if the measurements match the expected values. If the disk is moved to another computer or the boot chain is altered, the data cannot be read.

Example 2.1.2

On a computer with Secure Boot enabled, malware changes a few bytes of the boot loader file on disk. What happens at the next start-up?

1. UEFI runs and completes POST.
2. UEFI finds the boot loader in the EFI system partition and checks its signature before running it.
3. Because the file has changed, the computed value does not match the value in the signature. The signature cannot be verified.
4. UEFI does not run the boot loader and shows a security violation message. The malicious code never gets a chance to run.

Installing Windows 11 requires UEFI with Secure Boot support and TPM 2.0.

A closely related topic is how the disk is divided. A disk can be split into **partitions**, each used like a separate drive. A table at the start of the disk records where the partitions are. The format of this table is called the **partitioning scheme**, and two schemes are common.

Attribute	MBR	GPT
Firmware used with it	BIOS	UEFI
Largest disk	Up to 2 TB	Far above 2 TB
Number of partitions	At most 4 primary partitions	Many partitions

Table 2.1.4: Comparison of the MBR and GPT partitioning schemes.

MBR (Master Boot Record) is the older scheme. It stores locations on disk as 32-bit sector numbers. Since a sector is 512 bytes, the largest addressable space is calculated as follows:

> $2^{32} \times 512$ bytes = 4,294,967,296 × 512 = 2,199,023,255,552 bytes ≈ 2.2 TB

This number equals 2 TiB, that is, $2 \times 1{,}024^4$ bytes; this is where the "2 TB limit" comes from. **GPT** (GUID Partition Table) uses 64-bit numbers, so the limit effectively disappears. Table 2.1.4 compares the two schemes.

Example 2.1.3

Windows 11 will be installed on a 4 TB disk, and the disk will be split into five partitions for the system, programs, data, backup, and recovery. Which partitioning scheme should be chosen?

1. **Disk size:** 4 TB exceeds the roughly 2.2 TB limit of MBR. With MBR, nearly half of the disk (about 1.8 TB) would be unusable.
2. **Number of partitions:** five partitions exceed the MBR limit of four primary partitions.
3. **Operating system:** Windows 11 requires UEFI, and the scheme used with UEFI is GPT.

All three criteria give the same answer: GPT + UEFI.

2.1.3 The kernel

At the center of the operating system sits a small but critical part that has the authority to touch the hardware directly: the kernel.

2.1.3.1 The kernel and privileged execution

Definition 2.1.7

The **kernel** is the privileged core of the operating system. The processor can run in two modes: in **kernel mode** it can execute every instruction and access the hardware directly; in **user mode**, instructions that touch the hardware are forbidden. Applications run in user mode and request services from the kernel with a **system call**.

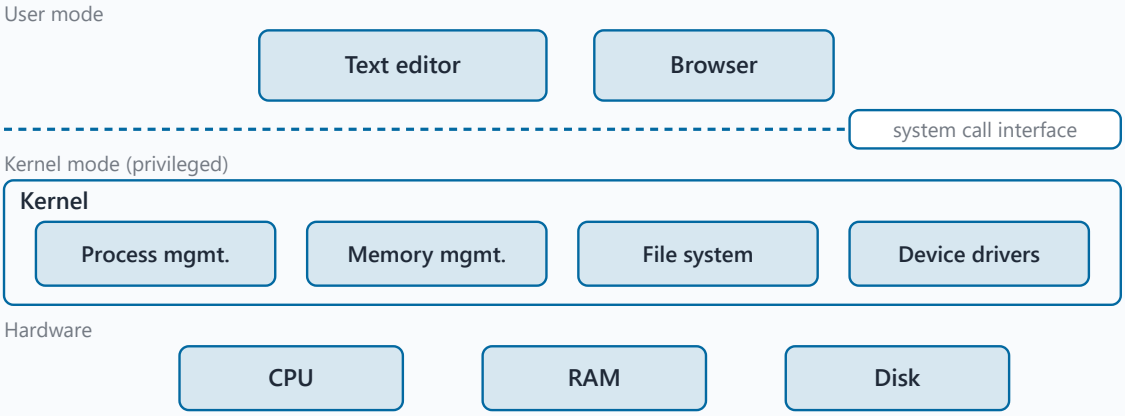


Figure 2.1.7: Applications in user mode (text editor, browser) reach kernel mode through the system call interface. The kernel's process, memory, file system, and device driver subsystems manage the hardware (CPU, RAM, disk).

Figure 2.1.7 draws the two modes as two floors. On the upper floor are the applications, with a boundary line beneath them. Applications can cross this line only through a single door, the system call interface. On the lower floor, the kernel's four subsystems talk to the hardware.

The reason for this separation is security and stability. If every program could write to the disk directly, a faulty program could overwrite another program's files and a malicious one could read other people's passwords. When the processor meets an instruction in user mode that touches the hardware, it does not execute it and hands control to the kernel. The kernel usually terminates that program. This way an application's bug affects only itself.

There is a cost too. A bug in the kernel can affect the whole system, because nothing above the kernel has the authority to stop it. The blue screen on Windows and the "kernel panic" message on Linux indicate an error that cannot be fixed at the kernel level. This is why every piece of code that enters the kernel, especially device drivers, is carefully tested and signed.

Common mistake

Thinking that "a program opened with an administrator account runs in kernel mode." An administrator account increases the *privileges* inside the operating system: changing system files, installing programs. But a program opened as administrator also runs in the processor's user mode and still reaches the hardware through system calls. Kernel mode is only for the code of the kernel and the drivers.

2.1.3.2 A system call step by step

Following how a system call works in a concrete example lets us see all the subsystems of the kernel together. A text editor wants to read the file *rapor.txt* on disk.

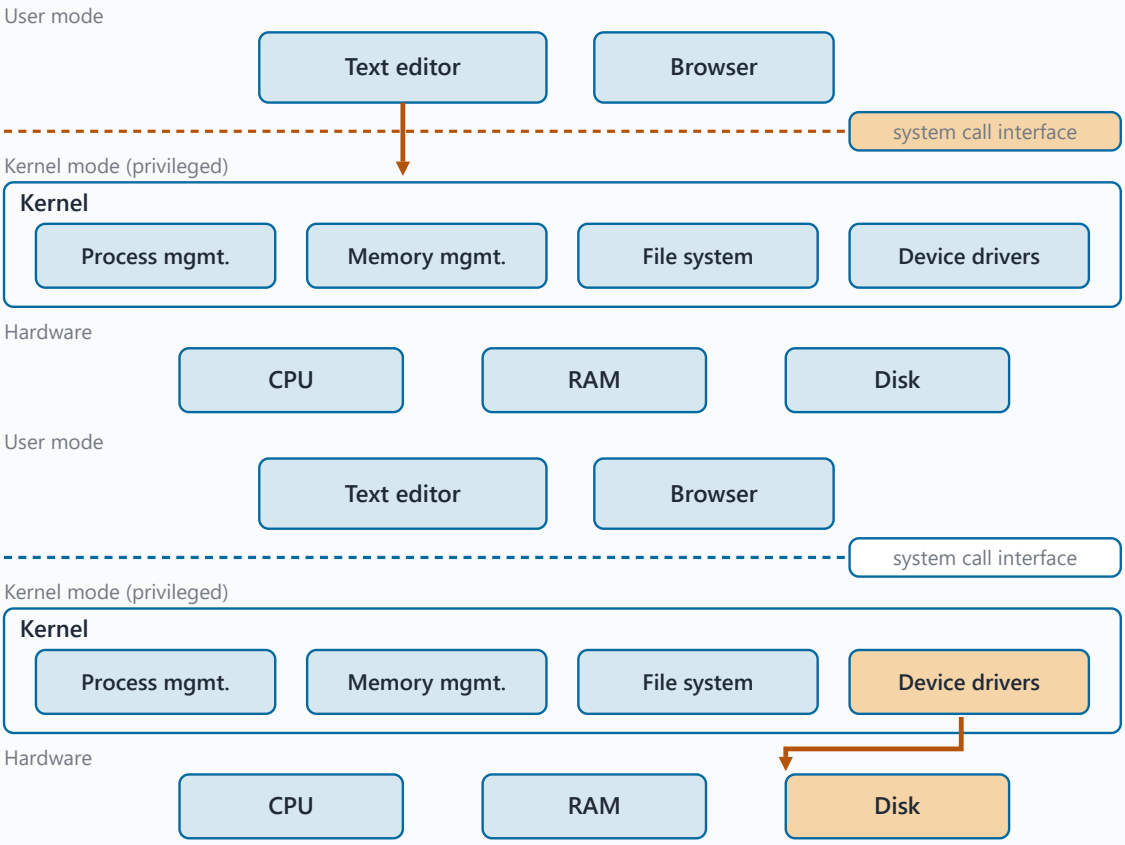


Figure 2.1.8: Two moments of a file-read system call. (a) The processor has switched from user mode to kernel mode; only kernel code runs now. (b) The disk driver has issued a read command to the disk; when the disk finishes, it notifies the processor with an interrupt.

Example 2.1.4

The text editor reads the file *rapor.txt*. In which mode is the processor at each step, and which subsystem runs?

Step	What happens?	Processor mode	Running part
1	The editor cannot reach the disk itself; it makes the `read` system call.	User	Editor
2	The processor switches to kernel mode; control is handed to the kernel.	Kernel	System call interface
3	The file's location on disk is found; the user's read permission is checked.	Kernel	File system
4	A read command is issued to the disk; when the disk finishes, it notifies with an interrupt.	Kernel	Device driver
5	The data that was read is copied into the editor's own memory region in RAM.	Kernel	Memory management
6	The call returns; the processor switches to user mode. The editor shows the data on screen.	User	Editor

The editor runs only in the first and sixth steps. All four steps in between are in kernel mode. The editor never sees the hardware directly.

Figure 2.1.8 shows two of these steps. The permission check in the third step shows the security value of the design: if the editor asks for a file it has no right to read, the call returns with an error code and the disk is never touched.

Two important concepts appear in the fourth step. The disk is very slow compared with the processor; an SSD read takes as long as tens of thousands of processor instructions. The processor does not sit idle during this time; it moves on to other work. When the disk finishes, it sends the processor a signal.

Definition 2.1.8

An **interrupt** is a signal by which a device or timer tells the processor about an event. When an interrupt arrives, the processor briefly drops what it is doing, runs the relevant code in the kernel, and then resumes where it left off.

Thanks to interrupts, the processor does not keep asking "is the disk done yet?" Pressing a key, a packet arriving from the network, and the timer's "time slice expired" notice are all interrupts too.

2.1.3.3 Kernel types

What to put inside the kernel is a design decision. Putting everything in the kernel is fast, because no mode switch is needed between subsystems. But the larger the kernel grows, the greater the chance of bugs inside it. Keeping the kernel small improves security, but frequent mode switches reduce speed. Between these two extremes, three designs are common.

Type	Idea	Example
Monolithic kernel	Process, memory, file system, and drivers in a single privileged kernel; fast but large	Linux (modules can be loaded)
Microkernel	Only the essentials in the kernel; drivers and file system in user mode; strong isolation	MINIX 3, QNX
Hybrid kernel	Takes the microkernel design but keeps some services in the kernel for speed	Windows NT kernel, macOS/iOS (XNU)

Table 2.1.5: Kernel types.

Table 2.1.5 compares the three designs. In a **monolithic kernel** all subsystems are one big program. Linux is built this way, but it allows drivers to be loaded as *modules* that can be added to and removed from the kernel when needed. In a **microkernel** the kernel does only the most basic work, such as interprocess communication, basic scheduling, and memory protection. Drivers and the file system run in user mode like ordinary programs. If a driver crashes, only that program is restarted; the system stays up. This is why QNX is common in cars and medical devices. A **hybrid kernel** sits between the two. Windows and Apple's systems follow this path.

The difference between a kernel and an operating system is best shown by Linux and Android. **Linux**, strictly speaking, is only a kernel. The "Linux" installed on a computer is in fact a **Linux distribution**: a package in which command-line tools, a desktop environment, and applications are added on top of the Linux kernel. Ubuntu, Fedora, and Debian are such distributions. **Android** also uses the Linux kernel, but the layers above it are entirely Android's own.

Layer	Role
Applications	Programs the user runs: messaging, maps, games
Android framework and runtime	Common services used by applications and the environment that runs application code
Linux kernel	Process, memory, driver, and power management
Hardware	Processor, touchscreen, camera, sensors

Table 2.1.6: The Android software stack (top to bottom).

Table 2.1.6 lists the layers of Android. A Linux desktop program does not run on an Android phone, and an Android app does not open directly on Ubuntu. Although the two share the same kernel, their upper layers differ.

2.1.4 Process management

A single-core processor executes the instruction of only one program at any given moment. Even so, it is possible to type while playing music on such a computer, because the operating system shares the processor among programs very quickly.

2.1.4.1 Concepts of process management

Process management is built on six concepts. The first three describe the things being managed, the last three the means of managing them.

Definition 2.1.9

A **program** is the whole of the instructions and data stored on disk. A **process** is a running instance of a program. A **thread** is a single flow of execution within a process. **Scheduling** is deciding which process gets the processor and when; the part of the kernel that makes this decision is called the **scheduler**. A **context switch** is saving the processor's state from one process and loading the state of another. **Concurrency** is the management of several tasks making progress in the same time interval.

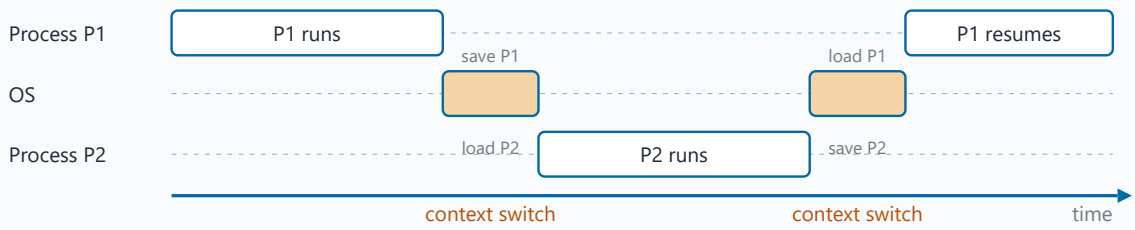


Figure 2.1.9: A context switch on a single CPU. P1 runs; the operating system saves P1's state and loads P2's state; P2 runs; another switch lets P1 continue where it left off.

Figure 2.1.9 shows the context switch on a time axis. A process's "context" is all the information the processor holds for that process: the values of the registers, the address of the next instruction, and the location of the process's memory table. When switching from P1 to P2, the operating system writes P1's context into a record in memory set aside for that process. It then loads P2's previously saved context into the processor. P2 continues from the instruction where it stopped, as if it had never paused.

During a context switch the processor advances no user program; it only does bookkeeping. A single switch takes a few microseconds, but when it happens thousands of times a second the loss becomes noticeable.

Common mistake

Thinking of concurrency as "running at the same time." On a single-core processor two processes never run at the same time; they run in turn, at very short intervals, and from the outside they appear to run simultaneously. Running truly at the same time is possible only with more than one core, and that is called *parallelism*.

2.1.4.2 Program, process, and thread

The difference between a program and a process is like the difference between a recipe and the dish cooking in a pot. Two dishes can be cooked in two separate pots from the same recipe; each pot has its own ingredients and its own stage. A program is a passive file sitting on disk; a process is a running instance of it in memory.

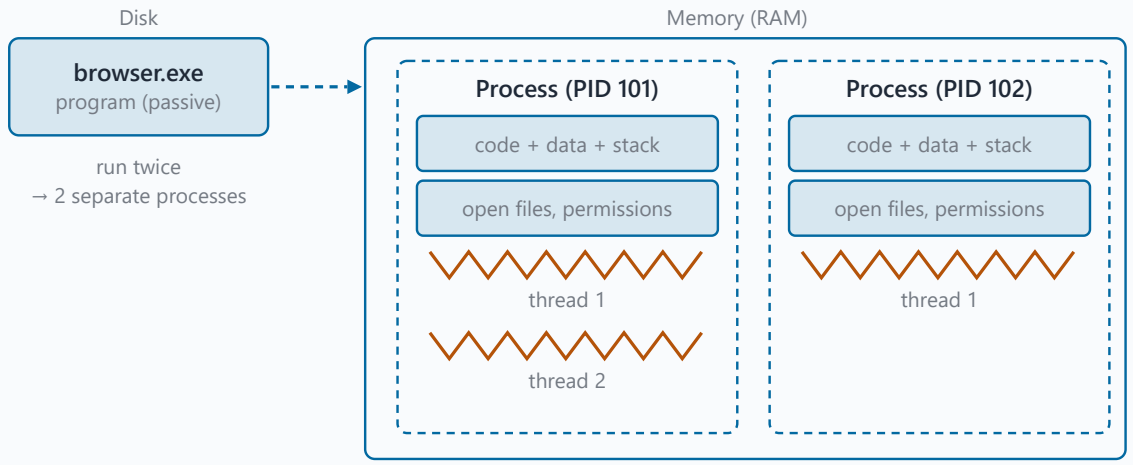


Figure 2.1.10: When the program *tarayici.exe* on disk is run twice, two separate processes appear in memory (PID 101 and PID 102). Each process has its own code, data, stack, open files, and permissions. The first process has two threads, the second one.

Figure 2.1.10 shows this relationship. The operating system identifies each process by a **process ID (PID)**. The PID is a number that does not change as long as the process lives. In Task Manager, or in the output of the `ps` command on Linux, each row is a process with its PID beside it.

Each process has a **separate memory region**. This region holds the program's code, a data area where variables are kept, and a stack that tracks function calls. The process also carries a list of the files it has opened and its permissions. Two processes cannot see each other's memory. When one browser window crashes, the other window usually keeps working; this is why modern browsers open each tab in a separate process.

Threads, by contrast, are flows of execution within the same process that run *sharing* that process's memory. A word processor can handle letters coming from the keyboard in one thread while running a spell check in the background in another. Both see the same document, that is, the same memory region. Sharing makes communication easy but needs care: if two threads try to change the same data at the same time, the data can be corrupted.

2.1.4.3 Process states

A process does not run all the time from the moment it is created until it ends. Sometimes it waits for its turn, sometimes for data to arrive from the disk. The operating system keeps the answer to "what state are you in right now?" for every process.

Definition 2.1.10

A **process state** is one of the following five. **New**: the process is being created. **Ready**: ready to run, waiting for its turn on the processor. **Running**: its instructions are executing on the processor right now. **Waiting**: waiting for an event, for example a disk read to finish or a key to be pressed. **Terminated**: it has finished its work. The list in which ready processes are lined up is called the **ready queue**.

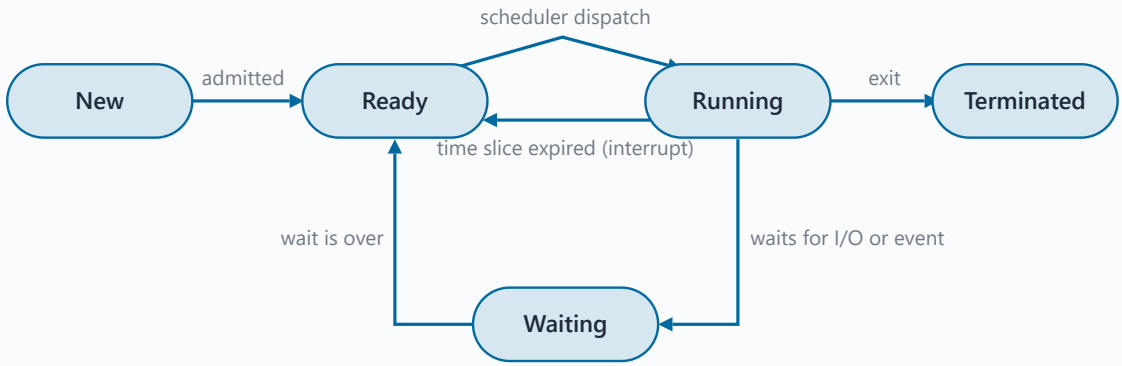


Figure 2.1.11: The process state diagram. Arrows show the allowed transitions and the cause of each: admitted, scheduler dispatch, time slice expired (interrupt), waits for I/O or event, wait is over, exit.

Figure 2.1.11 shows the five states and the six transitions between them. The transitions that are missing from the diagram are just as instructive as those that are present.

- **New → Ready** (admitted): the operating system allocates memory to the process and puts it in the ready queue.
- **Ready → Running** (scheduler dispatch): the process gets the processor.
- **Running → Ready** (time slice expired): the process's share of the processor ends; a timer interrupt hands control to the kernel and the process returns to the end of the queue.
- **Running → Waiting** (waits for I/O or event): the process asks for data from the disk, a key from the keyboard, or a packet from the network.
- **Waiting → Ready** (wait is over): the awaited event occurs and the process joins the queue again.
- **Running → Terminated** (exit): the process finishes its work.

Here I/O is the abbreviation for **input/output** operations. The diagram has no **Waiting → Running** arrow. A process whose wait is over does not get the processor directly; it first joins the ready queue and waits for its turn. There is no **Ready → Waiting** arrow either: a process can ask for something only while it is running, and therefore can move to waiting only while running.

On a single-core system, at most one process is in the Running state at any moment. The other ready processes wait in the queue. A process in the Waiting state is not even in the queue; the scheduler never picks it, because even if it got the processor there would be nothing it could do.

Common mistake

Confusing the Ready and Waiting states. Both mean "not running," but for different reasons. A ready process could run at once if given the processor; the only thing it lacks is the processor. A process in the Waiting state cannot make progress even if given the processor, because it is waiting for an external event.

2.1.4.4 State transitions step by step

Following the state diagram through the life of a single process makes concrete which events trigger the transitions.

Example 2.1.5

The user double-clicks a program and process P1 is created. P1 computes for a while, then reads data from disk, then computes again; meanwhile its time slice expires, and finally it finishes.

Step	Event	Transition	State of P1
1	The user double-clicks the program	—	New
2	The operating system admits the process and allocates memory	New → Ready	Ready
3	The scheduler picks P1 and its context is loaded	Ready → Running	Running
4	P1 asks for data from the disk	Running → Waiting	Waiting
5	The disk finishes and an interrupt arrives	Waiting → Ready	Ready
6	Its turn comes and it is picked again	Ready → Running	Running
7	The time slice expires, timer interrupt	Running → Ready	Ready
8	Picked again	Ready → Running	Running
9	It finishes and exits; its resources are reclaimed	Running → Terminated	Terminated

In nine steps P1 enters the Running state three times. Before every entry it always passes through the Ready state.

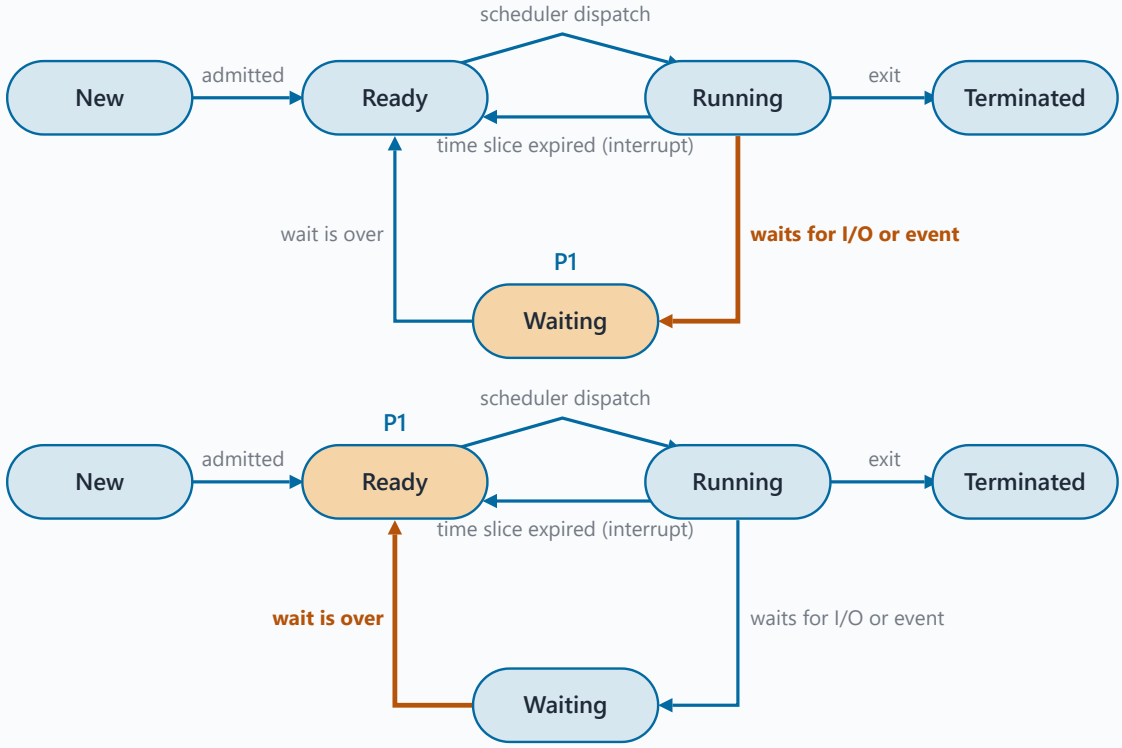


Figure 2.1.12: Two steps of Example 2.1.5. (a) Step 4: because P1 asks for data from the disk, it moves from Running to Waiting. (b) Step 5: when the disk finishes, P1 does not run at once; it first returns to the Ready state.

Figure 2.1.12 shows the two steps of the table that are most often confused. In the fourth step P1 gives up the processor of its own accord, because it has nothing to do until the data arrives. If it held the processor needlessly, other processes would be kept waiting too. In the fifth step the disk interrupt arrives, but at that moment another process may be running on the processor. P1 therefore returns to the queue. The transition in the seventh step does not happen at P1's request: the timer interrupt takes the processor away from P1 by force. This distinction is the basis of systems that can take the processor from a process by force.

2.1.4.5 Two processes, one CPU

When a process moves to the Waiting state, the processor becomes free. If the operating system fills that gap with another process, the total work finishes in less time. This idea is called **multiprogramming**. Its effect can be measured on a timeline.

Example 2.1.6

There are two processes on a single-core processor. P1 will first compute for 3 ms, then read from disk (the disk takes 3 ms), then compute for 2 ms more. P2 will compute for only 4 ms. Both are ready at t = 0; the scheduler picks P1 first. Each process gives up the processor only when it moves to waiting or when it finishes.

Time (ms)	CPU	Disk	P1	P2
0–3	P1	idle	Running	Ready
3	P1 asks to read, context switch		Waiting	Running
3–6	P2	P1's data	Waiting	Running
6	Disk finishes, interrupt arrives		Ready	Running
6–7	P2	idle	Ready	Running
7	P2 completes its 4 ms		Running	Terminated
7–9	P1	idle	Running	Terminated
9	P1 finishes		Terminated	Terminated

Both processes finish in 9 ms in total. Between 0 and 9 ms the processor is never idle: 3 ms of P1, 4 ms of P2, and 2 ms of P1 again, and *3 + 4 + 2 = 9*.

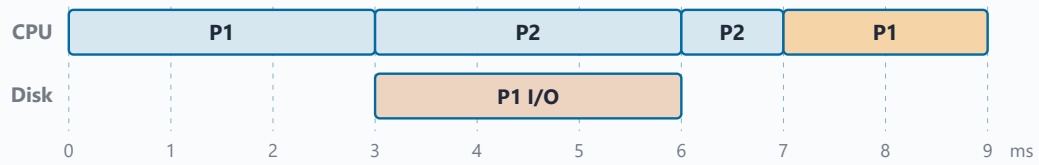


Figure 2.1.13: The timeline of Example 2.1.6. The top row shows the CPU, the bottom row the disk. Between 3 and 6 ms the CPU runs P2 while the disk reads P1's data; the two work in parallel.

Figure 2.1.13 makes the key point of the example visible: between 3 and 6 ms two separate pieces of hardware work at the same time. While the CPU runs P2, the disk does the read for P1. What would happen if the operating system did not exploit this overlap? If the CPU sat idle while P1 waited for the disk, that is, if the processes ran one after the other, the total time would be:

> 3 (P1 compute) + 3 (disk wait) + 2 (P1 compute) + 4 (P2 compute) = 12 ms

Multiprogramming saves 3 ms, and the total time drops from 12 ms to 9 ms. In sequential execution the CPU is working for only 9 of the 12 ms, so it is busy 75% of the time; with overlap that ratio is 100%. In real systems I/O waits are much longer, so the gain is much larger too.

There is one more detail in the example. P1's disk operation finishes at t = 6, but P1 runs only at t = 7. It spends the 1 ms in between in the ready queue, because P2 is on the processor and in this scheduling the running process is not taken off the processor by force.

2.1.4.6 Time sharing and round-robin scheduling

In the previous example, processes gave up the processor only when they moved to waiting. A process that does no I/O and runs a long computation could hold the processor for minutes under this arrangement; meanwhile the screen would not respond even if the user moved the mouse. Interactive systems therefore hand out the processor to processes in short slices.

Definition 2.1.11

Time sharing is distributing the processor among processes in rotation at short intervals. **Round-robin scheduling** gives each process in the ready queue, in turn, at most one **time slice (quantum)** of processor time. If the process does not finish within the slice, it goes to the end of the queue and a context switch takes place. The time slice is usually written with the letter q.

The rule of round-robin scheduling is simple: take the process at the head of the queue, run it for at most q, and if it is not finished, put it at the end of the queue. To see the consequences of the rule, the calculation has to be done step by step. For such calculations a **Gantt chart** is used: horizontal bars on a time axis showing which process the processor is on during each interval.

Example 2.1.7

Three processes are in the ready queue at t = 0, in the order P1, P2, P3. Their processor times are P1 = 5 ms, P2 = 3 ms, P3 = 2 ms. The time slice is $q = 2$ ms. Let us find each process's finish time.

Interval (ms)	CPU	What happens?	Remaining (P1, P2, P3)	Queue (after)
start	—	All three ready	5, 3, 2	P1, P2, P3
0–2	P1	Slice expires, P1 goes to the end	3, 3, 2	P2, P3, P1
2–4	P2	Slice expires, P2 goes to the end	3, 1, 2	P3, P1, P2
4–6	P3	P3 finishes (t = 6)	3, 1, 0	P1, P2
6–8	P1	Slice expires, P1 goes to the end	1, 1, 0	P2, P1
8–9	P2	P2 uses 1 ms, finishes (t = 9)	1, 0, 0	P1
9–10	P1	P1 uses 1 ms, finishes (t = 10)	0, 0, 0	idle

Finish times: P3 = 6 ms, P2 = 9 ms, P1 = 10 ms. Since the total work is $5 + 3 + 2 = 10$ ms, the last process finishes at t = 10.

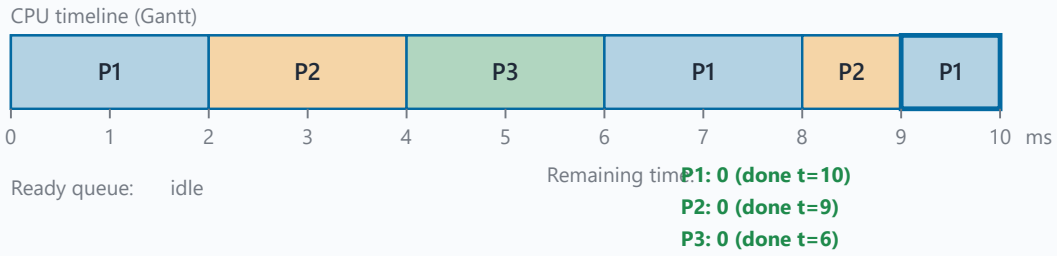


Figure 2.1.14: The completed Gantt chart of Example 2.1.7: P1, P2, P3, P1, P2, P1. The ready queue is empty; P3 finished at t = 6, P2 at t = 9, and P1 at t = 10.

Two details of the table need attention. First, the intervals 8–9 and 9–10 are one millisecond each. A process that finishes before its slice ends does not wait out the rest of the slice; the processor moves to the next one at once. Second, the queue column shows who goes to the end at each step, and this column determines the correctness of the calculation. If the queue order is skipped, all the following rows come out wrong.

Figure 2.1.14 gives the result as a chart. The chart has six bars, so the processor was given to a process six times. When two consecutive bars belong to different processes, a context switch falls between them; in this example there are 5 context switches. Thanks to the fast switches, all three processes seem to progress “at the same time.” In real systems the time slice is between a few milliseconds and a few tens of milliseconds; a person does not notice switches that short.

Choosing the time slice is a balance. As the slice shrinks, processes get turns more often, but the number of context switches grows. If the slice is chosen too large (*q = 5*), every process finishes on its first turn and short jobs wait behind the long one.

Time slice	Finish times (P1, P2, P3)	Context switches
q = 1	10, 8, 6	8
q = 2	10, 9, 6	5
q = 5	5, 8, 10	2

Table 2.1.7: Scheduling the same three processes (P1 = 5, P2 = 3, P3 = 2 ms) with different time slices.

Table 2.1.7 shows the effect with numbers; the duration of a context switch is taken as zero in these calculations.

Common mistake

Assuming that a process whose slice expires returns to the head of the queue. The process always goes to the *end* of the queue; otherwise the same process would never have given up the processor. The second common mistake is to give a full slice again to a process whose remaining time is shorter than the slice: in the interval 8–9 P2 has 1 ms left, and the processor

moves to P1 at t = 9.

2.1.4.7 Comparing scheduling algorithms

Round-robin scheduling is not the only option. In the process scheduling simulator, four algorithms can be compared. This section calculates three of them on the same example. Two measures are used for the comparison.

Definition 2.1.12

A process's **turnaround time** is the time from when it arrives in the ready queue to when it finishes: turnaround time = finish − arrival. **Waiting time** is the part of that time spent in the ready queue rather than running on the processor: waiting time = turnaround time − processor time.

The three algorithms work as follows. **FCFS** (first come, first served) runs processes in order of arrival, each to completion. **SJF** (shortest job first) picks, when the processor becomes free, the ready process with the shortest time and runs it to completion. **Round-robin** scheduling distributes the processor in slices, as in Example 2.1.7.

Example 2.1.8

Again P1 = 5, P2 = 3, P3 = 2 ms; all three arrived at t = 0, queue order P1, P2, P3. Let us find the average waiting time for each algorithm. Since arrival is 0, turnaround time equals the finish time.

1. **FCFS.** Schedule: P1 0–5, P2 5–8, P3 8–10. Finish times 5, 8, 10. Waiting: P1 = 5 − 5 = 0, P2 = 8 − 3 = 5, P3 = 10 − 2 = 8. Average $(0 + 5 + 8) / 3 = 13 / 3 \approx 4.33$ ms.
2. **SJF.** The shortest job is P3, then P2, then P1. Schedule: P3 0–2, P2 2–5, P1 5–10. Waiting: P3 = 0, P2 = 5 − 3 = 2, P1 = 10 − 5 = 5. Average $(0 + 2 + 5) / 3 = 7 / 3 \approx 2.33$ ms.
3. **Round-robin (q = 2).** Finish times P1 = 10, P2 = 9, P3 = 6. Waiting: P1 = 10 − 5 = 5, P2 = 9 − 3 = 6, P3 = 6 − 2 = 4. Average $(5 + 6 + 4) / 3 = 15 / 3 = 5$ ms.

Average waiting is lowest with SJF (≈ 2.33 ms) and highest with round-robin scheduling (5 ms).

Algorithm	Finish times (P1, P2, P3)	Average turnaround	Average waiting
FCFS	5, 8, 10	23 / 3 ≈ 7.67 ms	13 / 3 ≈ 4.33 ms
SJF	10, 5, 2	17 / 3 ≈ 5.67 ms	7 / 3 ≈ 2.33 ms

Algorithm	Finish times (P1, P2, P3)	Average turnaround	Average waiting
Round-robin (q = 2)	10, 9, 6	25 / 3 ≈ 8.33 ms	15 / 3 = 5 ms

Table 2.1.8: Results of the three algorithms on the same example (P1 = 5, P2 = 3, P3 = 2 ms; all at t = 0).

Table 2.1.8 puts the results side by side. The purpose of round-robin scheduling is not to minimize the average time but to leave no process without the processor for long. Under FCFS, P3 runs for the first time at t = 8; under round-robin it starts running at t = 4. In an interactive system, responding quickly to the user's click matters more than a slightly longer average time.

SJF minimizes the average waiting time, but it has two problems. First, how long a job will take is often not known in advance; the operating system can only estimate it from past behavior. Second, if short jobs keep arriving, a long job may never get its turn. A process that can wait forever for its turn is said to suffer *starvation*.

Common mistake

Thinking of waiting time as "the moment the process first starts running." Waiting time is the sum of *all* the intervals the process spends in the ready queue. In the round-robin example P1 runs during 0–2, then waits 4 ms during 2–6 and 1 ms during 8–9; the total waiting is 5 ms, which agrees with the formula 10 − 5.

2.1.4.8 Deadlock

Processes share not only the processor but also resources such as printers, files, or database records. Most of these resources can be used by only one process at a time. If two processes wrote to a printer at the same time, the pages would come out with the lines of two documents mixed together. For this reason a process "holds" a resource before using it and releases it when done. Under certain conditions, this arrangement can lead to processes waiting for one another forever.

Definition 2.1.13

A **deadlock** is a situation in which each process in a group is waiting for a resource held by another process in the group, and none of them can make progress.

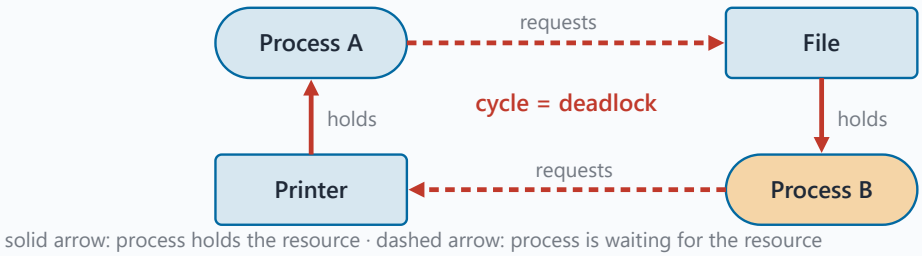


Figure 2.1.15: A resource allocation graph. A solid arrow shows that a resource is held by a process, a dashed arrow that the process is waiting for that resource. A holds the printer and waits for the file, B holds the file and waits for the printer; the arrows form a cycle.

Example 2.1.9

Process A will print a report and, while printing, write an entry to a file. Process B will update the file and then get a summary from the printer. The events occur in this order:

1. A takes the printer and uses it.
2. B locks the file and uses it.
3. A requests the file without releasing the printer. The file is held by B, so A waits.
4. B requests the printer without releasing the file. The printer is held by A, so B waits.

The waiting chain closes as A → File → B → Printer → A. Since neither releases what it holds, both wait forever.

Figure 2.1.15 shows the situation after the fourth step. The visible sign of a deadlock in a graph is a cycle in the arrows. Every process in the cycle holds one resource and waits for the next resource in the cycle.

A deadlock arises only if four conditions hold at the same time. These conditions were defined in 1971 by Coffman and colleagues; deadlock is still explained by these four conditions today.

Theorem 2.1.1 (The four conditions of deadlock)

For a deadlock, *all* of the following four conditions are necessary:

1. **Mutual exclusion:** a resource can have only one owner at a time.
2. **Hold and wait:** a process waits for another resource without releasing the one it holds.
3. **No preemption:** a resource cannot be taken from its owner by force; it becomes free only when the owner releases it.
4. **Circular wait:** the waiting processes are linked to one another in a ring.

If even one of the conditions is broken, a deadlock cannot occur.

Theorem 2.1.1 also points to the solution: breaking any one of the four conditions is enough. In practice the easiest condition to break is circular wait. It is made a rule that all processes request resources *in the same order*, for example the printer first, then the file. If B in Example 2.1.9 also has to request the printer first, then while the printer is with A, B waits without holding anything. A also gets the file, finishes its work, releases both resources, and B continues. The ring cannot form.

Breaking mutual exclusion is often not possible: a printer is by nature single-owner.

2.1.5 Memory management

The processor reads instructions and data only from RAM; so every program has to be loaded into memory to run. The operating system must find room for each process, keep processes from entering one another's space, and offer a way out when RAM is not enough.

2.1.5.1 Tasks of memory management

Role	Meaning
Allocation	Providing each program with a memory region according to its needs
Protection	Preventing one process from changing another process's memory without permission
Virtual memory	Giving programs an address space independent of physical RAM
Paging	Managing memory in fixed-size pieces
Swapping	Temporarily moving some data to disk when necessary
Sharing	Opening common code or data to several processes in a controlled way

Table 2.1.9: The six tasks of memory management.

Table 2.1.9 lists the tasks. The first two, allocation and protection, are essential to every memory manager. Virtual memory and paging are today's way of doing these two tasks flexibly. Swapping comes into play when RAM is not enough. Sharing serves to use memory efficiently. For example, five

processes started from the same browser keep the program's code in RAM once, not five times; sharing is safe because the code is only read. Each process's own data, however, stays separate.

2.1.5.2 Allocation, protection, and swapping step by step

The basic ideas of memory management show up clearly in a model simpler than real systems. In this model each process is given a single *contiguous* region in RAM, and the region is described by two numbers.

Definition 2.1.14

The **base register** holds the starting address of the process's region in RAM, and the **limit register** holds the length of the region. The process works with **logical addresses** that treat the start of its own region as 0. On every access the hardware compares the logical address with the limit. If the address is less than the limit, **physical address = base + logical address**; otherwise the access is blocked.

These two registers are inside the processor, and the hardware uses them automatically on every memory access. When the process changes, the operating system loads the new process's base and limit values into the registers; this is part of the context switch. A process cannot change these registers itself, because the instructions that write to them run only in kernel mode.

Example 2.1.10

RAM is 640 KB, and the first 100 KB is used by the operating system. Processes A (200 KB) and B (150 KB) are loaded in turn, A makes two memory accesses, and then process C arrives asking for 250 KB. Addresses are in KB.

Step	Event	Calculation	Result
1	Start	640 − 100 = 540	0–100 operating system, 540 KB free
2	A is loaded	base = 100, limit = 200; 100 + 200 = 300	A: 100–300; 340 KB free
3	B is loaded	base = 300, limit = 150; 300 + 150 = 450	B: 300–450; 640 − 450 = 190 KB free
4	A requests address 150	150 < 200 ✓; 100 + 150 = 250	Goes to physical address 250
5	A requests address 250	250 ≥ 200 ✗	Access blocked; control passes to the kernel
6	C asks for 250 KB	190 < 250	No room; the waiting B is written to disk, leaving 640 − 300 = 340 KB of contiguous free space

Step	Event	Calculation	Result
7	C is loaded	base = 300, limit = 250; 300 + 250 = 550	C: 300–550; 640 − 550 = 90 KB free; B on disk

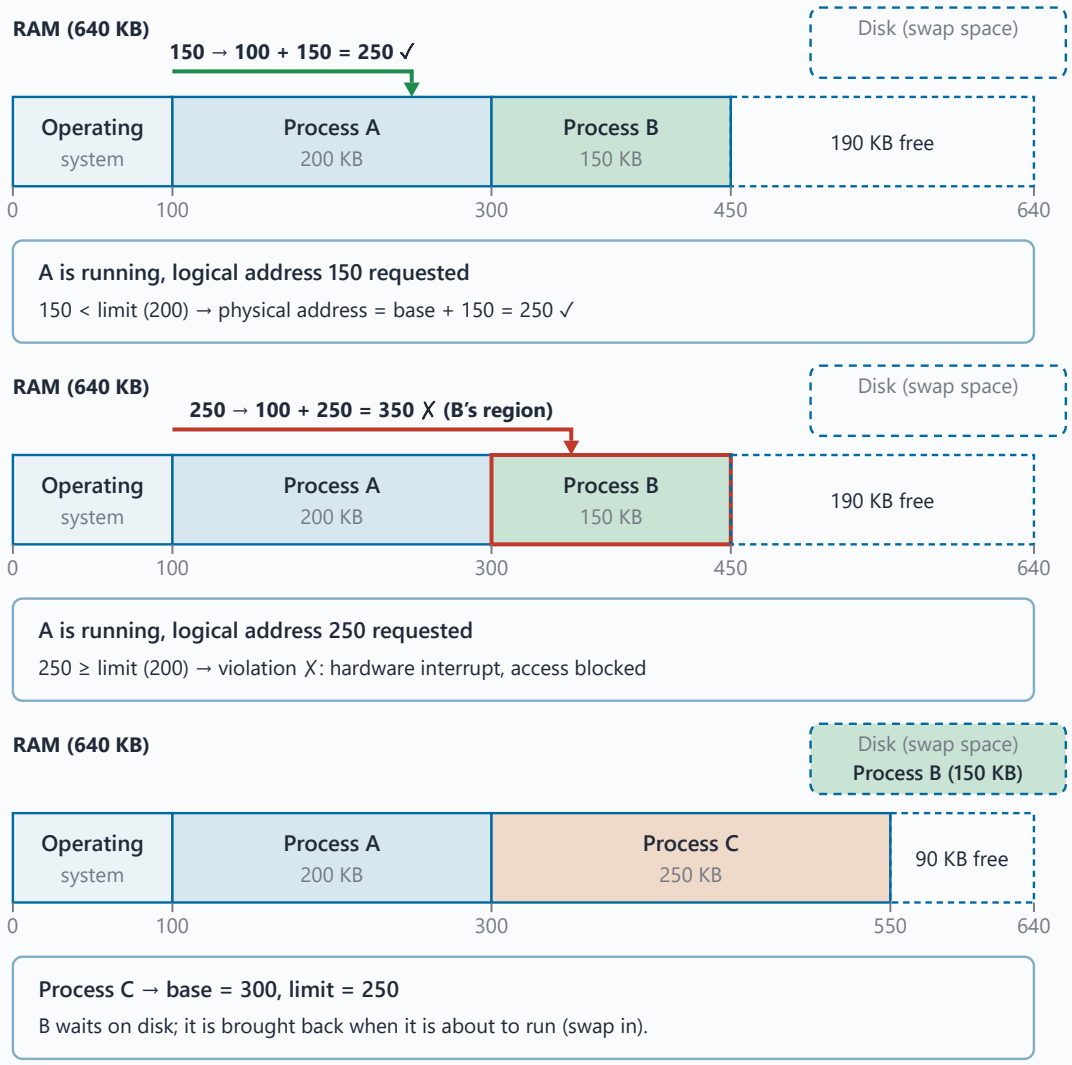


Figure 2.1.16: Three steps of Example 2.1.10. (a) Step 4: A's address 150 is below the limit and is translated to physical address 250. (b) Step 5: address 250 exceeds the limit; if allowed, it would reach B's region at 350. (c) Step 7: B has been written to disk and C is loaded at 300–550.

Figure 2.1.16 shows the important moments of the table on a strip of RAM. In allocation (steps 1–3) each new process takes its place at the start of the free area, and the regions do not overlap. In a valid access (step 4) A asks for address 150 from its own point of view; A does not know that its region starts at 100 in RAM. The hardware first compares 150 with the limit, then adds the base: 100 + 150 = 250.

Protection violation (step 5). This time A asks for address 250. 250 is not less than 200; the address is outside A's region. The hardware does not perform the access; it raises an interrupt and hands control to the kernel. In a real system the operating system would terminate A with an "access violation" (segmentation fault) error; to continue the example, A's region is left in place here. Had the hardware not blocked it, the physical address would have been 100 + 250 = 350, that is, a write into B's region. Thanks to protection, B's memory is not corrupted.

Swapping (steps 6–7). C asks for 250 KB. RAM has 190 KB free between 450 and 640; since 190 < 250, C does not fit. At that moment B is waiting for a disk operation and will not run for a while. The operating system writes all of B's memory to the **swap space** on disk. This operation is called *swapping out*. A contiguous free area of 340 KB appears between 300 and 640, and C is loaded at 300–550, leaving 90 KB. When B is about to run again, it is brought back into RAM. If there is no room, another process is written to disk this time.

Definition 2.1.15

Swapping is temporarily writing the memory of a process, or part of it, to disk to free up RAM, and reading it back when needed. The area set aside on disk for this is called the **swap space**.

Swapping increases capacity but has a heavy price: the disk is much slower than RAM. The next section puts numbers on this difference.

Common mistake

Thinking that the limit register holds the region's *end address*. The limit is the length of the region. For A the limit is 200, not an end address of 300. This is why the comparison is made with the logical address (150 < 200), not with the physical address. Valid logical addresses run from 0 to 199; even address 200 is outside the region.

This simple model has a weakness: as processes come and go, small gaps appear in RAM, and even when the total free space is enough, a contiguous piece may not be found. Virtual memory and paging solve this problem.

2.1.5.3 Virtual memory and paging

Definition 2.1.16

Virtual memory is the arrangement that gives each process its own **virtual address** space, starting at 0 and independent of physical RAM. In **paging**, the virtual address space is divided into equal-sized **pages** and physical RAM into **frames** of the same size. Each process has a **page table**, which records whether each page is in a frame or on disk. A common page size is 4 KB (4,096 bytes).

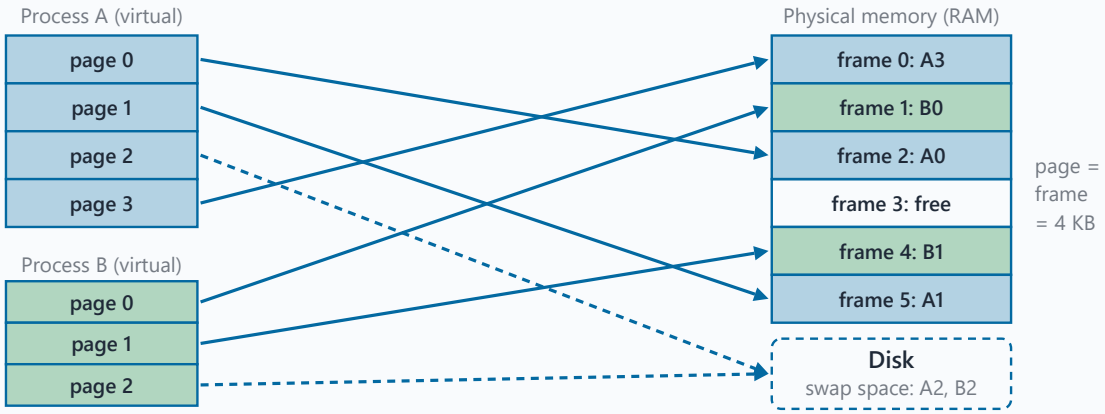


Figure 2.1.17: The virtual pages of two processes are mapped to physical frames or to disk. A's pages 0, 1, and 3 are in frames 2, 5, and 0 respectively, and its page 2 is in the swap space on disk. B's pages 0 and 1 are in frames 1 and 4, and its page 2 is on disk. Frame 3 is free.

Figure 2.1.17 shows three important consequences of paging.

- **Each process sees its own world.** Both A and B start at page 0. The "address 0" of the two processes is in different places in RAM: A's in frame 2, B's in frame 1. A process cannot generate an address into another's page at all, because that frame is not in its own table. Protection comes automatically.
- **Pages need not be consecutive in RAM.** A's pages are scattered across frames 2, 5, and 0. Every free frame fits any page; the problem of searching for contiguous space disappears.
- **Rarely used pages can be moved out to disk.** A2 and B2 wait on disk. They take no room in RAM until the process needs them. Swapping is now done not with the whole process but page by page.

The swap space is called the **page file** on Windows and *swap* on Linux and macOS. The price of virtual memory is speed. A RAM access takes about 50–100 nanoseconds (ns); a read from an SSD takes about 0.1 milliseconds (ms). Let us compare the two times by converting them to the same unit:

> 0.1 ms = 100,000 ns; 100,000 ÷ 100 = 1,000 and 100,000 ÷ 50 = 2,000

Reading a page from disk takes 1,000 to 2,000 times longer than one RAM access. As long as swapping is rare, this difference is not noticed. But if RAM is badly insufficient, the operating system starts to write and read pages constantly, the disk light flickers, and the computer slows down noticeably. This is why a shortage of RAM causes slowness even when the disk has plenty of free space.

2.1.5.4 Address translation step by step

When a process generates a virtual address, the **memory management unit (MMU)** inside the processor translates it into a physical address. Translation has two stages: first the address is split into a *page number* and an *offset* within the page, then the page number is looked up in the table.

> page = virtual address ÷ page size (integer quotient); offset = virtual address mod page size

> physical address = frame × page size + offset

Here "mod" means the remainder of the division. The offset never changes in translation: because the page is moved to a frame as a whole, byte 4 of the page is also byte 4 of the frame. All that changes is that the frame number takes the place of the page number.

Example 2.1.11

The page size is 4,096 bytes. The page table of process A: page 0 → frame 2, page 1 → frame 5, page 2 → on disk, page 3 → frame 0. Frame 3 is free. A first reads virtual address 4100, then 8200.

1. **Division.** 4100 ÷ 4096 = 1, remainder 4. That is, 4100 = 1 × 4096 + 4: page 1, offset 4.
2. **Table.** Page 1 → frame 5. The page is in RAM.
3. **Physical address.** 5 × 4096 + 4 = 20,480 + 4 = 20,484. The read is made from this address.
4. **Division.** 8200 ÷ 4096 = 2, remainder 8. That is, 8200 = 2 × 4096 + 8: page 2, offset 8.
5. **Table.** Page 2 is not in RAM but on disk. The MMU raises a **page fault**; the instruction is halted and the operating system steps in.
6. **Loading.** The operating system reads page 2 from disk into the free frame 3 and updates the table: page 2 → frame 3.
7. **Retry.** 3 × 4096 + 8 = 12,288 + 8 = 12,296. This time the translation succeeds.

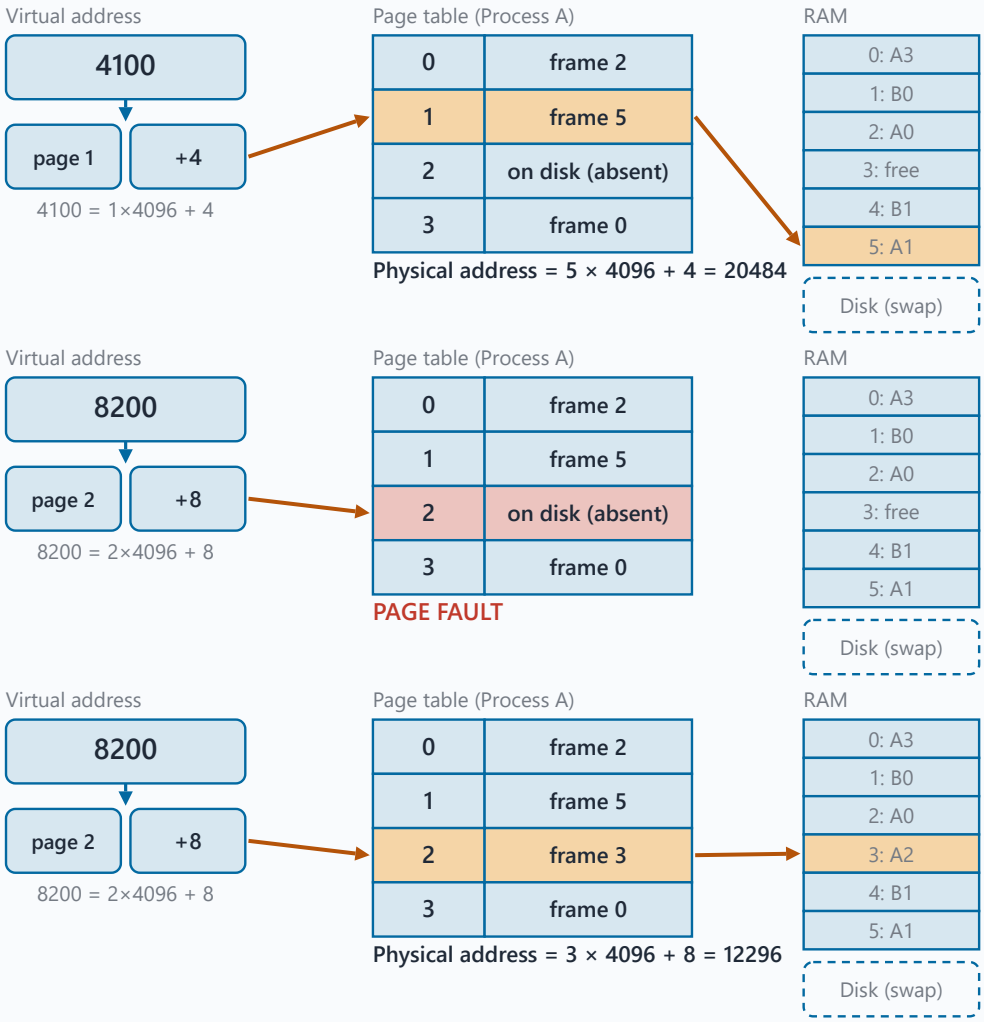


Figure 2.1.18: Three moments of Example 2.1.11. (a) 4100 = 1 × 4096 + 4; page 1 is in frame 5, physical address 20484. (b) The page of address 8200 (page 2) is on disk: page fault. (c) Page 2 has been loaded into frame 3; physical address 3 × 4096 + 8 = 12296.

Definition 2.1.17

A **page fault** is the MMU's report that the page the process accessed is not in RAM at that moment. The operating system loads the page from disk into a frame, updates the page table, and re-executes the interrupted instruction.

Figure 2.1.18 shows three moments of the translation. Despite its name, a page fault is mostly not a programming error; it is the normal operation of virtual memory. When a program starts, most of its pages are still on disk and arrive in RAM one by one through page faults on their first accesses. The program never notices; only that access takes a little longer. A real error appears only when the process accesses an address that does not belong to it and is not in the table at all; then the operating system terminates the process with an access violation.

Division and remainder become even easier with binary numbers. Since $4096 = 2^{12}$, the *last 12 bits* of a virtual address give the offset and the *remaining bits* give the page number. The hardware does no division; it just splits the bits. In hexadecimal each digit is 4 bits, so the last three hexadecimal digits are the offset:

Virtual address	Hexadecimal	Page · offset	Frame	Physical address (hexadecimal)	Physical address
4100	1004_{16}	1 · 004	5	5004_{16}	20484
8200	2008_{16}	2 · 008	3	3008_{16}	12296

Table 2.1.10: Address translation in hexadecimal: the last three digits (12 bits) are preserved unchanged.

Table 2.1.10 shows this shortcut. 4100 is 1004_{16} in hexadecimal. The leading 1 is the page number, and the trailing 004 is the offset. Since page 1 is in frame 5, the leading 1 is replaced by 5: 5004_{16} = 20484. The decimal calculation gave the same result.

Common mistake

Adding the offset directly to the frame number when finding the physical address: 5 + 4 = 9. The frame number is a sequence number; to get the frame's starting address it must be multiplied by the page size: 5 × 4096 = 20,480. The offset is added to that start.

2.1.5.5 Page replacement: LRU and FIFO

When a page fault occurs and there is a free frame, the job is easy. If RAM is full, the operating system has to move one page out to disk and load the new one in its place. Deciding which page to evict is called **page replacement**. A wrong choice causes a new page fault for the same page right afterwards. Every fault also means a slow disk read.

Definition 2.1.18

The **LRU** (least recently used) rule evicts the page whose last use is the oldest. The **FIFO** (first in, first out) rule evicts the page that was loaded into RAM first. When the requested page is already in RAM, it is called a **hit**.

The idea behind LRU is this: programs tend to reuse soon the pages they used recently. A loop runs the same instructions again and again; a function accesses the same variables again and again. A page that has not been touched for a long time is less likely to be needed soon.

Example 2.1.12

The process has only 3 frames in RAM. The process requests pages in the order 7, 0, 1, 2, 0, 3, 0, 4. How many page faults occur under the LRU rule?

Request no.	Page	Result	Evicted	Pages in RAM (last use)
1	7	fault	—	7 (1)
2	0	fault	—	7 (1), 0 (2)
3	1	fault	—	7 (1), 0 (2), 1 (3)
4	2	fault	7	2 (4), 0 (2), 1 (3)
5	0	hit	—	2 (4), 0 (5), 1 (3)
6	3	fault	1	2 (4), 0 (5), 3 (6)
7	0	hit	—	2 (4), 0 (7), 3 (6)
8	4	fault	2	4 (8), 0 (7), 3 (6)

Of the 8 requests, 6 are page faults and 2 are hits. The first three faults are loads into free frames; at each of the next three faults, a page is evicted.

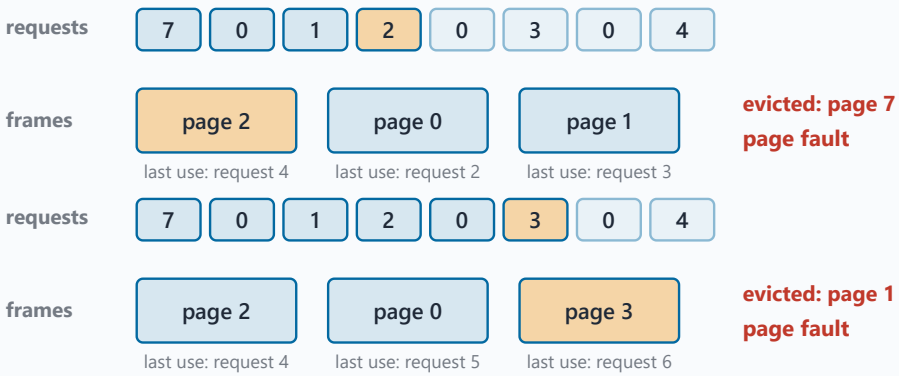


Figure 2.1.19: Two eviction moments of Example 2.1.12. (a) Request 4: there is no room for page 2; page 7, with the oldest last use (request 1), is evicted. (b) Request 6: for page 3 the oldest use is on page 1 (request 3); page 0 is kept because it was used just now, at request 5.

Figure 2.1.19 shows two of the eviction decisions. The numbers in parentheses in the table are the information LRU keeps for its decision: the request at which each page was last used. The page to evict is the one with the smallest number. When page 0 is a hit at the fifth request, only its last-use time is updated, from 2 to 5. This small update saves page 0 from being evicted at the sixth request.

What happens if the same request sequence is processed with FIFO? FIFO looks only at the order in which pages entered RAM; hits do not change the order.

Request no.	Page	Result	Evicted	RAM (oldest to newest)
1–3	7, 0, 1	3 faults	—	7, 0, 1
4	2	fault	7	0, 1, 2
5	0	hit	—	0, 1, 2
6	3	fault	0	1, 2, 3

Request no.	Page	Result	Evicted	RAM (oldest to newest)
7	0	fault	1	2, 3, 0
8	4	fault	2	3, 0, 4

Table 2.1.11: Processing the same sequence (7, 0, 1, 2, 0, 3, 0, 4; 3 frames) with FIFO. The RAM column shows the order of entry: the leftmost is the oldest.

Table 2.1.11 shows that FIFO gives 7 page faults, one more than LRU. The difference arises at the sixth request. FIFO does not take into account that page 0 was just used; because 0 is the page that entered earliest, it evicts it. Right after that, the seventh request asks for page 0 again, and an unnecessary fault occurs. LRU had kept the frequently used page 0.

If each page fault requires an SSD read of about 0.1 ms, LRU's 6 faults cost about 0.6 ms and FIFO's 7 faults 0.7 ms; hits finish within nanoseconds.

LRU is not better than FIFO on every sequence, but it gives fewer faults on most real programs. Exact LRU is expensive because it needs a timestamp on every access; operating systems use cheaper rules that approximate it.

Common mistake

Taking LRU and FIFO to be the same because both evict an "old" page. FIFO evicts the page whose *entry into RAM* is the oldest, LRU the page whose *last use* is the oldest. A hit changes nothing in FIFO, while in LRU it refreshes the page's last-use time.

2.1.6 Device and file management

The user talks to the computer through the keyboard, screen, printer, and disks. The operating system manages these devices with drivers and organizes the raw data on the disk as files that have a name, a location, and an owner.

2.1.6.1 Devices and drivers

A laser printer and an inkjet printer expect completely different commands for the same "print page" request. Graphics cards from two manufacturers have different registers. If the operating system held all these differences inside itself, the operating system would have to change for every new device.

Definition 2.1.19

A **device driver** is software that translates the operating system's common requests into operations specific to a particular piece of hardware. The driver is usually written by the device's manufacturer and is added to the operating system later.

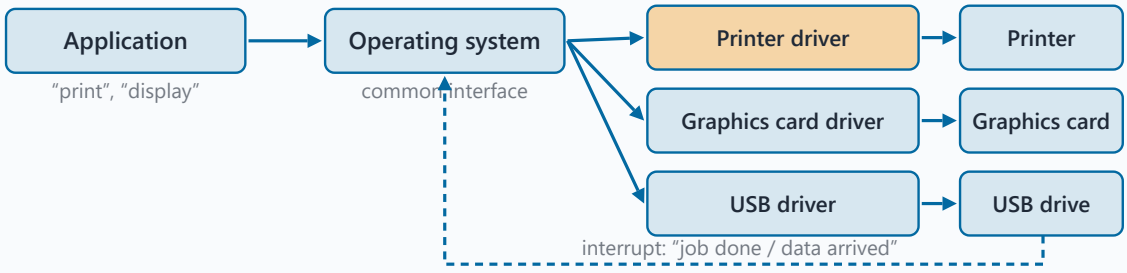


Figure 2.1.20: The application issues common requests such as "print" or "display." The operating system passes them through a common interface to the right driver (printer, graphics card, USB). When the device finishes, it reports "job done / data arrived" with an interrupt.

Figure 2.1.20 shows where drivers sit: between the operating system and the hardware, a separate interpreter for each device. Three more concepts of device management are part of this arrangement.

- **Plug and play:** the operating system detects a newly attached compatible device, reads its identity, and loads and configures the right driver automatically. This is why a USB drive appears in File Explorer within a few seconds of being plugged in.
- **Interrupt:** a device reports an event to the processor. A key press on the keyboard, a printer finishing a job, and a packet arriving from the network are all interrupts.
- **Buffer:** a memory area that temporarily holds data between components of different speeds.

Example 2.1.13

A 20-page document is sent to a printer that prints 10 pages per minute. Without a buffer, the word processor would wait for the printer on every page and look frozen for 2 minutes. With a buffer, the operating system puts all 20 pages into the print queue quickly and the word processor is freed at once. The driver sends the pages from the buffer at the printer's speed; when each page is done, the printer reports that it is ready with an interrupt. Printing still takes 2 minutes, but the user keeps working in the meantime.

2.1.6.2 The file system

A disk on its own is a large store made of millions of numbered blocks. No one can remember something like "my homework starts at block 18,452." Users and programs refer to files by name. The file system builds the bridge between the name and the block number.

Definition 2.1.20

A **file system** is the arrangement that defines how data is named, organized, and located on a storage device. A **file** is a unit of data with a name and contents. A **folder** (directory) groups files and subfolders. A **path** specifies a file's location in the folder structure. **Metadata** is a file's properties other than its contents: size, date, owner, and permissions. **Permissions** determine the rights to read, write, and execute. **Journaling** helps the file system stay consistent after unexpected interruptions.

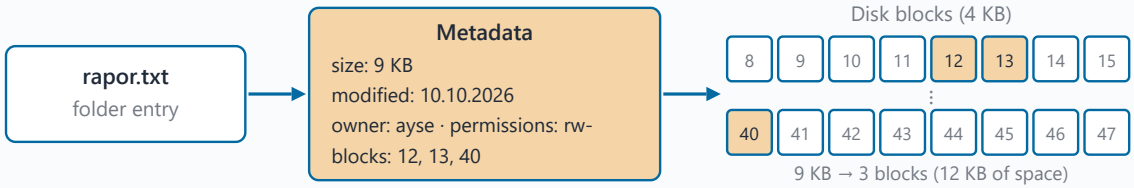


Figure 2.1.21: The *rapor.txt* entry in the folder points to the file's metadata: size 9 KB, modified 10.10.2026, owner ayse, permissions rw-, blocks 12, 13, and 40. The disk is divided into 4 KB blocks; a 9 KB file takes 3 blocks, that is, 12 KB of space.

Figure 2.1.21 shows how a file is found on disk as a chain of three links. The folder holds the name of each file in it and the location of that file's metadata record. The metadata record states all the file's properties and which disk blocks hold the contents. The contents themselves are in those blocks. The blocks need not be consecutive: the first two pieces of *rapor.txt* are in blocks 12 and 13, and the third piece is in block 40.

Example 2.1.14

The block size is 4 KB. How many blocks and how much space does the 9 KB *rapor.txt* take on disk? How much space does a 1-byte file take on the same disk?

1. 9 ÷ 4 = 2.25. Since there are no fractional blocks, round up: 3 blocks.
2. Space used: 3 × 4 = 12 KB. In the last block 12 − 9 = 3 KB is left empty; this space cannot be given to another file.
3. A 1-byte file also needs at least one block: it takes 4 KB of space.

This is why a file's Properties window in Windows shows two separate values, "Size" and "Size on disk."

Journaling means the file system writes what it is about to do to a separate journal area before making a change. Saving a file takes several steps: the contents are written to blocks, the metadata is updated, and the free-block list changes. If the power is cut in the middle of these steps, the file system may be left inconsistent: for example, a block may appear both free and in use. A journaling file system checks the journal at start-up and either completes or rolls back the unfinished work. This way a long disk scan is not needed.

2.1.6.3 File system types

Each operating system family has developed its own file system. Which computers can read and write a USB drive depends on the file system it was **formatted** with. Formatting means writing an empty file system structure to a partition, and it erases the partition's old contents.

File system	Platform	Journaling	When is it used?
NTFS	Windows (macOS read-only)	Yes	Windows system disk; when large files, permissions, and encryption are needed
FAT32	Almost every device and operating system	No	Older devices, cameras, simple memory cards; a single file can be at most 4 GB
exFAT	Windows, macOS, and current Linux	No	USB drives and SD cards with large files; moving data between platforms
ext4	Linux	Yes	Linux system and data disks
APFS	macOS, iOS, iPadOS	No classic journal; consistency through copy-on-write	Default file system of Apple devices

Table 2.1.12: Common file systems.

Table 2.1.12 compares five common file systems. The most practical limit in the table is FAT32's file size limit. FAT32 stores a file's size as a 32-bit number. The largest number that 32 bits can write is $2^{32} - 1 = 4{,}294{,}967{,}295$; so a single file can be at most one byte short of 4 GiB. APFS's "copy-on-write" method, on the other hand, writes changed data not over the old data but to a new place; when the operation completes, it switches the pointer to the new place. If an interruption occurs, the old version stays intact.

Example 2.1.15

A 3 GB film and a 5 GB video (1 GB = 10^9 bytes) are to be copied to a FAT32-formatted USB drive. The limit is 4,294,967,295 bytes. Since 3,000,000,000 bytes is below the limit, the film is copied; since 5,000,000,000 bytes exceeds the limit, the video cannot be copied even if

there is room on the drive. If the drive will be written on both Windows and macOS, exFAT is chosen, because macOS can only read NTFS. Since formatting erases the contents, a backup is made first.

2.1.6.4 The file system tree and paths

Because folders can contain folders, the file system has the shape of a tree. At the top of the tree sits the **root** folder. Folders are the branches of the tree and files are its leaves.

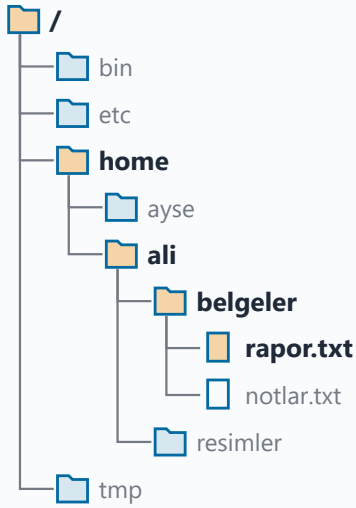


Figure 2.1.22: A Linux file tree. The path from the root folder (/) to *rapor.txt* is highlighted: / → home → ali → belgeler → rapor.txt.

Figure 2.1.22 shows the tree on a Linux system. The root has the folders *bin*, *etc*, *home*, and *tmp*. Users' personal folders are under *home*. The highlighted path shows how a file's address is formed by writing folder names one after another.

Definition 2.1.21

An **absolute path** starts at the root and gives the file's full address: `/home/ali/belgeler/rapor.txt`. A **relative path** is written relative to the **current directory**, that is, the folder you are in at that moment. `..` denotes the parent folder and `.` the current folder itself.

Example 2.1.16

In the tree of Figure 2.1.22, how is *rapor.txt* reached by a relative path from three different current directories?

1. **Current directory /home/ali:** `belgeler/rapor.txt` . Go down one folder.
2. **Current directory /home/ali/belgeler:** `rapor.txt` . The file is already here.
3. **Current directory /home/ayse:** `../ali/belgeler/rapor.txt` . First go up to the parent folder (home), then down into ali and belgeler.

All three relative paths point to the same file, and the absolute equivalent of each is `/home/ali/belgeler/rapor.txt` .

On Windows the same idea is written with two differences. There is not a single root; each drive has its own root, and a drive is shown with a letter (C:, D:). The folder separator is a backslash instead of a forward slash: `C:\Users\Ali\Documents\rapor.txt`. On Linux and macOS names are case-sensitive (*Rapor.txt* and *rapor.txt* are two different files); on Windows by default they are not.

2.1.6.5 Opening and reading a file step by step

When a program reads a file, it makes a series of system calls to the kernel in the background: first it opens the file, then reads it piece by piece, and finally closes it. These three calls show how the file system and process management work together.

Definition 2.1.22

A **file descriptor (fd)** is the small integer the kernel gives a process for an open file. In later read and write calls the process refers to the file by this number, not by its name. In every process, 0 is open from the start for standard input, 1 for standard output, and 2 for standard error. On Windows the same idea is called a **handle**.

Example 2.1.17

An editor opens the 150-byte file `/home/ali/rapor.txt` and reads it to the end in pieces of 100 bytes. What happens at each call?

Call	What the kernel does	Return value	Position
`open`	Resolves the path piece by piece: / → home → ali → rapor.txt. Checks the metadata: size 150 bytes, read permission present. Creates an open-file record.	fd = 3	0
`read` (100)	Reads bytes 0–99	n = 100	100
`read` (100)	Only 50 bytes remain; reads 100–149	n = 50	150
`read` (100)	No bytes left to read	n = 0 (end of file)	150

Call	What the kernel does	Return value	Position
`close`	Deletes the record; number 3 becomes free again	0 (success)	—

The editor gets 100 + 50 = 150 bytes with three `read` calls. The third call returning 0 tells the program that it should stop reading.

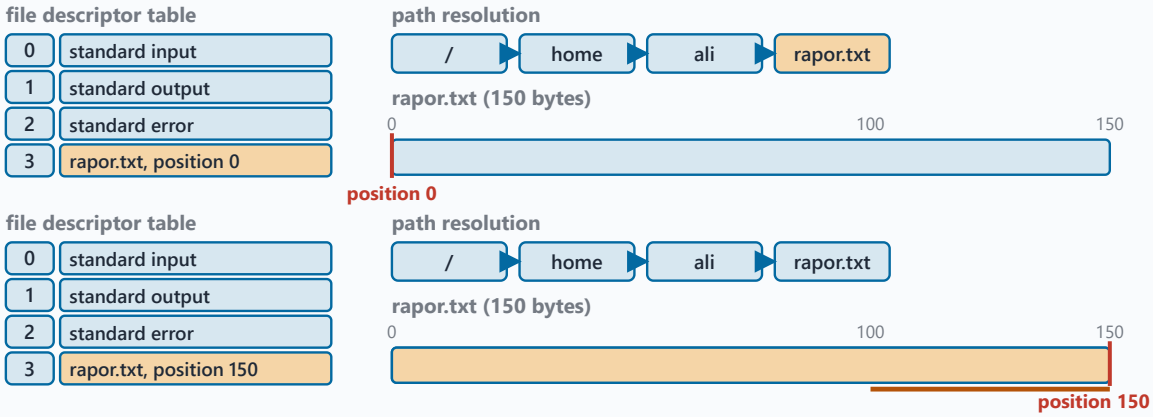


Figure 2.1.23: Two moments of Example 2.1.17. (a) After `open`: entry 3 in the file descriptor table points to *rapor.txt*, position 0. (b) After the second `read`: the position has reached 150, the end of the file.

Figure 2.1.23 shows the file descriptor table and the position in the file. Let us follow the table step by step.

Path resolution. In the `open` call the kernel resolves the path one component at a time from the start. First it reads the root folder and looks for the name *home* in it. When it finds it, it reads the *home* folder and looks for *ali*. Each folder is a list of the names in it and where those names are on disk. Finally *rapor.txt* is found and its metadata is examined. If the user has no read permission, `open` returns an error and the file is not opened. On later reads the path does not have to be resolved again; this work is done only at open time.

File descriptor and position. The kernel gives the process the *smallest* unused number; since 0, 1, and 2 are taken, the first file opened gets 3 and the second gets 4. Each `read` advances the position automatically by the number of bytes it read; the application never knows block numbers.

Closing. `close` deletes the record and the number becomes free. When a process ends, the operating system itself closes any files it left open. But if a long-running server program does not close the files it opens, the number of descriptors allowed per process eventually runs out and the program can no longer open new files.

2.1.7 User interfaces

The place where the user meets the operating system is the interface: windows, gestures on a touchscreen, or a command line.

2.1.7.1 Interface types

Definition 2.1.23

A **graphical user interface (GUI)** is based on windows, icons, menus, and interaction with a mouse. A **command-line interface (CLI)** is an interface in which the user types text commands and sees the results as text. A **touch interface** works with taps, swipes, and finger gestures.

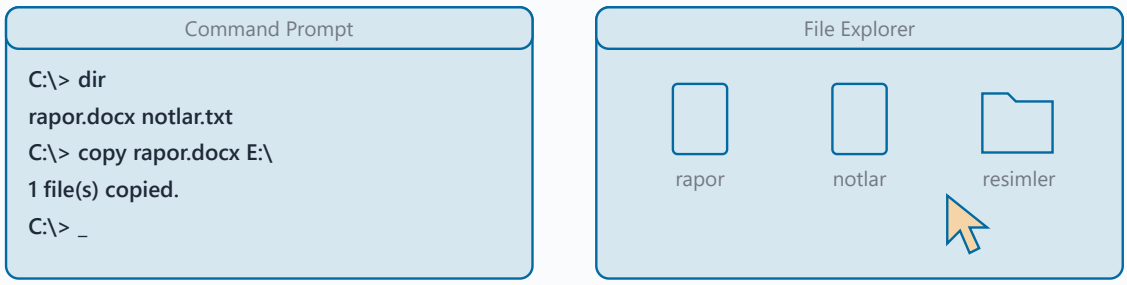


Figure 2.1.24: The same job in two interfaces. On the left, in Command Prompt, `dir` lists the folder and `copy rapor.docx E:\` copies the file. On the right, File Explorer shows the same folder with icons.

Figure 2.1.24 shows the same folder in two interfaces. In the graphical interface the files appear as icons and the file is dragged to copy it. At the command line, the contents of the folder are listed with the `dir` command and the file is copied with the `copy` command. Both ways end in the same system calls; the difference is only in how the user conveys the request.

A graphical interface is easy to learn: menus show what can be done. The command line is fast, repeatable, and suited to automation. Renaming 500 photos one by one with the mouse takes hours; at the command line a short script finishes the job in seconds and can be run automatically every night. This is why most servers have no graphical interface installed at all.

Every interface is also supported by **accessibility** tools: a screen reader reads the text aloud, a magnifier enlarges the screen, and a high-contrast theme makes text stand out.

2.1.7.2 Command-line basics

At the command line each command is one line: first the name of the command, followed by *arguments* separated by spaces. In the command `copy a.txt b.txt`, `copy` is the name of the command and `a.txt` and `b.txt` are the arguments. Commands always run in the current directory; relative paths are interpreted relative to that folder. The shell on Linux and macOS and Command Prompt (cmd) on Windows use different command names for the same jobs.

Task	Linux / macOS	Windows (cmd)
Show the current folder	`pwd`	`cd` (no arguments)
List the contents of a folder	`ls`	`dir`
Change folder	`cd folder`	`cd folder`
Create a folder	`mkdir new`	`mkdir new`
Copy a file	`cp a.txt b.txt`	`copy a.txt b.txt`
Delete a file	`rm a.txt`	`del a.txt`

Table 2.1.13: Equivalents of basic commands on Linux/macOS and Windows (cmd).

Table 2.1.13 compares the six most common tasks in the two worlds. `pwd` takes its name from "print working directory." `cd` means "change directory." Using commands and paths together makes it possible to move around the file tree.

Example 2.1.18

On Linux the current directory is `/home/ali`. In the tree, *belgeler* and *resimler* are under `/home/ali`, and *notlar.txt* is under `/home/ayse`. How does the current directory change as the following commands are run in order?

Command	Explanation	Next current directory	Output
`cd belgeler`	Relative path: one folder down	/home/ali/belgeler	—
`cd ../resimler`	One up (ali), then resimler	/home/ali/resimler	—
`cd ../../ayse`	Two up (ali, home), then ayse	/home/ayse	—
`ls`	Lists the contents	/home/ayse	notlar.txt
`cd /tmp`	Starts with "/": absolute path	/tmp	—
`pwd`	Prints the current directory	/tmp	/tmp

Each `..` goes one folder up. A path that starts with `/` starts from the root no matter what the current directory is.

Common mistake

Thinking that a file deleted with `rm` or `del` goes to the Recycle Bin. These commands delete the file directly; there is no easy way to get it back. Before running the command, make sure that the current directory and the file name are correct.

2.1.7.3 File permissions and chmod

On Linux and macOS, each file's permissions are kept separately for three groups of users: the file's **owner**, the **group** the owner belongs to, and **others**. For each group there are three permissions: **r** (read), **w** (write), and **x** (execute). A file's permissions are thus written with nine characters: `rw-r--r--`. A dash in place of a letter means that permission is absent.

Permissions are changed with the `chmod` command and are usually given as a three-digit number. Each digit stands for one group. The value of a digit is found by adding the weights of the permissions in that group: r = 4, w = 2, x = 1.

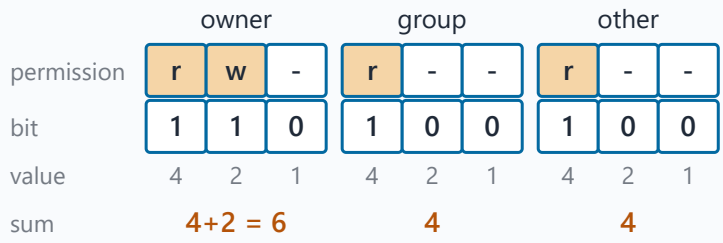


Figure 2.1.25: `chmod 644` : owner rw- (110 → 4 + 2 = 6), group r-- (100 → 4), other r-- (100 → 4).

Figure 2.1.25 shows where 644 comes from. Since the weights are 4, 2, and 1, each group's three permissions are really a three-bit binary number. The owner's permission rw- is binary 110, and 110_2 = 6. r-- is 100_2 = 4. Since three bits can express eight values from 0 to 7, each digit lies between 0 and 7.

Digit	Binary	Permission	Meaning
7	111	rwx	Read, write, execute
6	110	rw-	Read, write
5	101	r-x	Read, execute
4	100	r--	Read only
0	000	---	No permissions

Table 2.1.14: Commonly used permission digits.

Example 2.1.19

(a) What are the permissions after the command `chmod 755 yedekle.sh`? (b) A log file should be readable and writable only by its owner, with no one else having access. Which number should be used? (c) Which number writes the permission `rw-rw-r--`?

1. **(a)** 7 = 4 + 2 + 1 → rwx; 5 = 4 + 1 → r-x; 5 → r-x. The result is `rwxr-xr-x`: the owner can do everything, while the group and others can read and execute but not modify.
2. **(b)** Owner: rw- = 4 + 2 = 6. Group: --- = 0. Other: --- = 0. The number is `600`.
3. **(c)** rw- = 6, rw- = 6, r-- = 4. The number is `664`.

Three groups with three permissions each make nine bits in total; so there are $2^9 = 512$ different permission combinations. Since each digit has 8 values, the same number comes out as $8 \times 8 \times 8 = 512$.

Common mistake

Reading permission digits like a decimal number: "644 is greater than 600, so it grants more permissions." Each digit stands for a separate group. With 644, the group and others can read; with 600 they cannot. Also, giving `chmod 777` just because a problem is not getting solved makes the file modifiable by everyone and is a serious security hole.

2.1.8 User and privilege management

A computer can be used by several people; thousands of people can connect to a server. The operating system asks two questions at every operation: who is performing the operation, and does this person have the right to do it? These two questions are answered in two separate stages.

Definition 2.1.24

Authentication checks that the user is who they claim to be: "Who?" It is done with a password, a PIN, or biometric data such as a fingerprint. **Authorization** determines what the authenticated user may do: "What can they do?" Permissions and account type are checked at this stage.

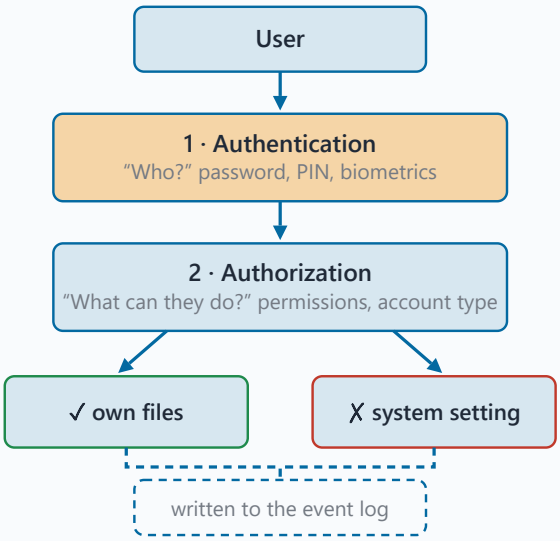


Figure 2.1.26: The authentication and authorization flow. A standard user can open their own files (✓) but cannot change a system setting (✗). Allowed and denied accesses are written to the event log.

Figure 2.1.26 shows the two stages in order. The user first proves their identity. After that, an authorization check is made separately for every operation. In the example, the same user can open their own file but cannot change a system setting. Both results are written to the event log.

Accounts are generally of two kinds. A **standard account** is for everyday use and limits changes that affect the system. An **administrator account** can make system-wide changes such as installing programs, changing system settings, or accessing other users' files. On Windows even an administrator account shows a confirmation window before such an operation; on Linux the same operation is done by typing `sudo` before the command, and a password is requested.

Definition 2.1.25

The **principle of least privilege**: each user and each program is given only as much authority as needed to do its job, and no more.

The value of this principle shows up during an attack. Malware in an e-mail attachment opened with a standard account runs only with that user's privileges. It can harm the user's own files, but it cannot change system files and cannot reach other users. If the same attachment were opened on the computer of someone who always works with an administrator account, it could take over the whole system.

Logging allows important events to be examined afterwards: if hundreds of failed password attempts appear in the log at midnight, the administrator learns of the attack.

Example 2.1.20

A student with a standard account signs in with a password, opens their own homework file, then tries to change the system clock and is blocked. At which stage is each step?

1. **Signing in:** the password is correct → authentication succeeds.
2. **Opening the homework file:** the file's owner is the student and read permission exists → authorization succeeds (✓).
3. **Changing the clock:** a system-wide setting is closed to the standard account → authorization denied (✗); written to the event log.

The obstacle is in authorization, not in authentication. The student has proven who they are; they simply lack the privilege for that one operation.

2.1.9 Types of operating systems

A smartphone, a web server, and a car's brake unit use operating systems that do the same basic jobs, but their priorities differ: battery life stands out on the phone, uninterrupted operation on the server, and timely response in the brake unit.

2.1.9.1 Personal computer systems

On desktop and laptop computers, four families stand out.

- **Windows:** the system with the widest range of hardware and applications. It runs on computers from many manufacturers; most games and business software are written for Windows first.
- **macOS:** runs only on Apple computers. Because the same company makes the hardware and the software, the two are tightly integrated.
- **Linux distributions:** can be customized for different desktop environments and purposes. Most are free and can run well even on older hardware.
- **ChromeOS:** focuses on web- and cloud-based use. Because most applications and data live on the internet, it starts quickly even on low-spec devices; offline use is limited.

2.1.9.2 Mobile operating systems

There are two systems in the mobile world. **Android** is a broad ecosystem that runs on devices from different manufacturers. **iOS and iPadOS** are the integrated systems Apple developed for its phones and tablets. The priorities of mobile systems differ from the desktop: a touch interface, power management that extends battery life, sensors such as location and acceleration, and constantly changing wireless connections. Applications running in the background are tightly restricted, because every running process drains the battery.

Definition 2.1.26

A **sandbox** is an isolated runtime environment in which an application can access only its own data. Access to resources such as location, camera, microphone, and contacts has to pass through the operating system's permission check and often through the user's approval.

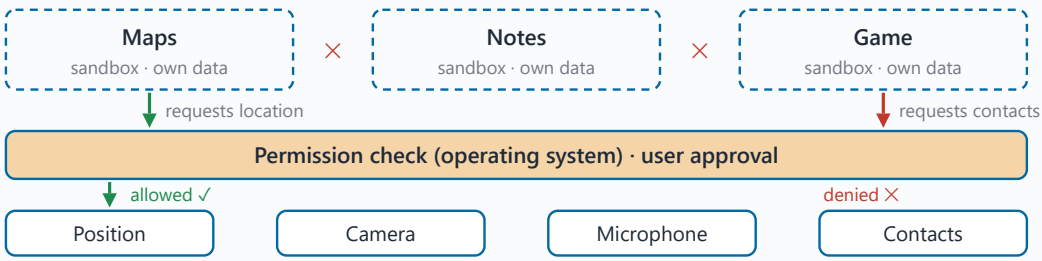


Figure 2.1.27: The maps, notes, and game apps run in their own sandboxes and cannot reach each other's data (✗). The maps app asks for location and the game asks for the contacts list; the requests pass through the operating system's permission check and the user's approval.

Figure 2.1.27 shows application isolation. Each application lives in its own box. The notes app cannot read the maps app's saved addresses. When an application wants to reach the phone's shared resources, the operating system steps in and asks the user. A maps app asking for location is an expected request, and the user grants it. A game asking for the contacts list is suspicious, and the user can refuse. This way a malicious app cannot collect personal data without the user noticing.

2.1.9.3 Server operating systems

A server is a computer that provides services to other computers over a network. Server operating systems are shaped by this job.

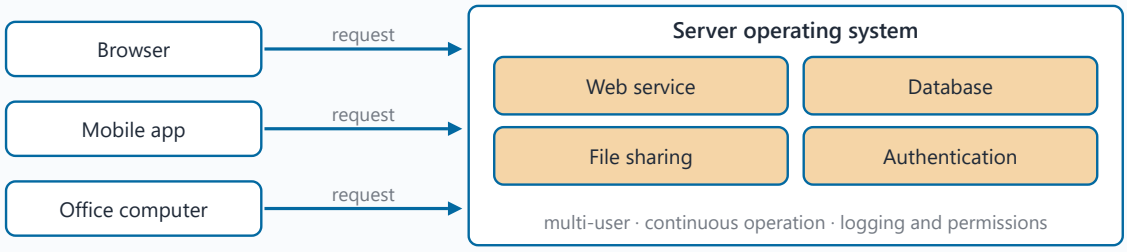


Figure 2.1.28: Requests from clients such as a browser, a mobile app, and an office computer go to the web, database, file sharing, and authentication services of the server operating system. The server is multi-user and runs continuously; it keeps logs and checks permissions.

Figure 2.1.28 shows that a server serves many clients at the same time. The priorities of server operating systems are as follows:

- **Multi-user operation:** managing many users and services at the same time.
- **Network services:** web, database, file sharing, and authentication services.
- **Continuity:** months of uninterrupted operation and remote management. When a server restarts, all the services that depend on it are interrupted.
- **Security:** permission, logging, updating, and network control.
- **Automation:** repeatable management with the command line and configuration tools. Instead of configuring a hundred servers by hand one by one, the same script is run on all of them.

Common examples are Linux distributions, Windows Server, and Unix derivatives. The great majority of web servers on the internet use Linux.

2.1.9.4 Embedded and real-time systems

Definition 2.1.27

An **embedded operating system** is an operating system that runs with limited processor, memory, and energy for a specific device task; it is found in routers, car control units, and medical devices. A **real-time operating system** aims for an operation to produce not just the correct result but the result within a set time, that is, before the **deadline**.

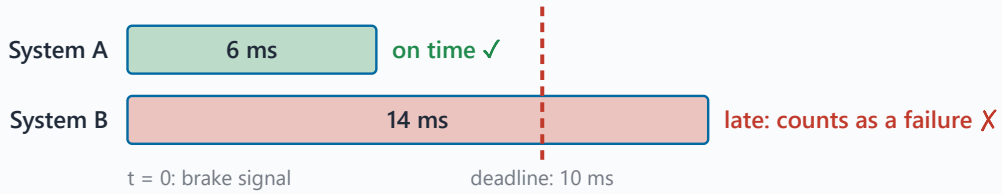


Figure 2.1.29: A deadline in a real-time system. The brake signal arrives at t = 0 and the deadline is 10 ms. System A responds in 6 ms (on time ✓); system B responds in 14 ms (late: counts as a failure ✗).

Figure 2.1.29 shows the basic rule of a real-time system: a correct result that arrives late is also wrong. Both systems produce the right brake command, but B is counted as failed because it misses the deadline. In a desktop operating system no one notices a window opening 4 ms late. In a braking system the same delay means distance.

Example 2.1.21

In a car traveling at 100 km per hour, the brake signal arrives at t = 0. System A responds after 6 ms, system B after 14 ms. How far does the car travel in these times?

1. Convert the speed to meters per second: 100 km/h = 100,000 m ÷ 3,600 s ≈ 27.78 m/s.
2. System A: 27.78 × 0.006 ≈ 0.17 m.

3. Deadline 10 ms: 27.78 × 0.010 ≈ 0.28 m.
4. System B: 27.78 × 0.014 ≈ 0.39 m.

B misses the deadline by 4 ms; before braking begins, the car travels about 0.39 − 0.17 = 0.22 m farther than with A. A real-time system must guarantee that it will not exceed 10 ms even in the worst case.

In real-time systems being fast on average is not enough; the worst-case response time must also be known and bounded. This is why real-time schedulers interrupt a less important task at once when an urgent one arrives.

Common mistake

Using the phrase “real-time” to mean “very fast.” A real-time system does not have to be very fast; it has to be on time. A system with a deadline of 1 second that always meets it is real-time. A system that responds in 1 ms on average but can occasionally lag by 50 ms is not.

2.1.9.5 Choosing an operating system

When choosing an operating system for a school laboratory, a software company, or a factory, no single criterion is enough. There is no such thing as “the best operating system”; there is only the one that best fits the requirements.

Requirement	Criterion to evaluate
Application	Availability of the required software and its versions
Hardware	Driver and processor architecture support
Security	Update period, privilege model, and management tools
Usability	Interface, accessibility, and learning needs
Cost	The sum of license, support, training, and maintenance

Table 2.1.15: Criteria to evaluate when choosing an operating system.

2.1.10 Licensing

When software is bought, what is bought is often not the software itself but the right to use it under certain conditions. The license sets these conditions.

Definition 2.1.28

A **license** is the legal text that sets out which rights the software's owner gives the user and under which conditions. **Source code** is the form of a program, written in a programming language, that a person can read and modify.

2.1.10.1 Open-source software

In **open-source** software the source code is open to everyone, and the license grants rights in four areas.

- **Source code:** can be examined within the terms of the license.
- **Modification:** can be adapted to need if the license allows.
- **Distribution:** the original or a modified version can be shared; the license sets the terms.
- **Community:** development and support can be carried out together by individuals, institutions, and companies.

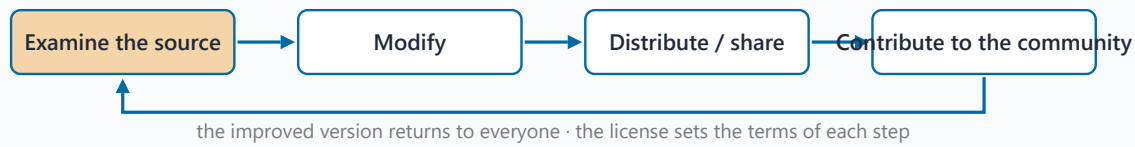


Figure 2.1.30: The open-source cycle: examine the source, modify, distribute, and contribute to the community. The improved version returns to everyone; the license sets the terms of each step.

Figure 2.1.30 shows how open source grows as a cycle. A developer examines the code, fixes a bug or adds a feature, and sends the change back. If the change is accepted, it reaches everyone with the next release. The Linux kernel has grown this way, with the contributions of thousands of developers.

Open-source licenses fall into two main groups. Licenses such as the **GPL** (GNU General Public License) require that a modified version, if distributed, be distributed under the same license and together with its source code; this approach is called *copyleft*. Permissive licenses such as **MIT** and **Apache** allow the software to be used even inside a closed product; they generally ask only that the original license text and copyright notice be kept.

Common mistake

Equating "open source" with "free of charge" and "without support." The license of an open-source program is generally free of charge, but many companies sell paid support, training, and maintenance for open-source products. Conversely, not every program distributed free of charge is open source: a free game's source code may be closed. What matters is not the price but the rights the license grants.

2.1.10.2 Commercial and closed-source software

In **closed-source (proprietary)** software the source code stays with the developing organization. The user is given only the compiled, executable program.

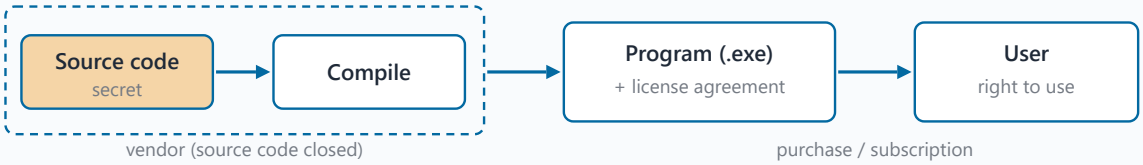


Figure 2.1.31: The closed-source model. The vendor keeps the source code secret and compiles it; the compiled program (.exe) is given to the user under a license agreement. The user obtains only the right to use it, by purchase or subscription.

Figure 2.1.31 shows this model. Four features of commercial software stand out.

- **Right to use:** the software is not bought; what is usually obtained is a license to use it under certain conditions. The conditions are written in the **end-user license agreement (EULA)**: how many devices it can be installed on, whether it can be used in business, whether reverse engineering is forbidden.
- **Source code:** usually kept closed within the developing organization. The user cannot fix a bug themselves; they wait for the vendor's fix.
- **Support:** official support and service-level agreements can be offered. For example, how many hours it takes to respond when a problem is reported is guaranteed by contract.
- **Cost:** there may be license, subscription, maintenance, and upgrade fees.

2.1.10.3 Comparing license models

Criterion	Open source	Proprietary (closed source)
Source access	Open under the license	Generally closed
Customization	Possible within the scope of the license	Limited to the options the vendor offers
Support	Community or paid provider	Vendor or authorized partner
Cost	The license can be free; running costs remain	A license or subscription may be required
License example	GPL, MIT, Apache	End-user license agreement
Product example	Linux, Firefox, LibreOffice	Windows, Microsoft Office, Adobe Photoshop

Table 2.1.16: Comparison of open-source and proprietary (closed-source) license models.

Table 2.1.16 compares the two models on six criteria. Being commercial and being open source are separate dimensions: commercial products such as Red Hat Enterprise Linux can also be open source. Both models have strengths. Open source is transparent because the code can be examined by

anyone, and it reduces dependence on a single vendor. Closed source offers a single responsible vendor and contracted support. Many organizations use both: Linux on servers, Windows and Office on the desktop.

Example 2.1.22

A student club writes a program using an MIT-licensed library and another GPL-licensed library, and wants to distribute the program. What obligations arise?

1. **MIT library:** the program can be distributed under any license desired. Only the library's copyright notice and license text must be included with the distribution.
2. **GPL library:** if a program containing GPL code is distributed, the whole program must be offered under the GPL together with its source code.
3. **Result:** if the club wants to keep the program's source code closed, it cannot use the GPL library; it should look for a permissively licensed alternative.

Being "open source" does not mean there are no obligations. The license text determines the rights and obligations in every case.

2.1.11 Updates and support

An operating system does not stay as it was on the day it was installed. New security holes are found, hardware changes, and users want new features. The vendor answers these with updates.

Concept	Meaning
Security update	Closes known vulnerabilities
Feature update	Brings new functions and interface changes
Driver update	Can improve hardware compatibility or stability
Support period	The time span in which the vendor provides updates
Backup	Provides a way back before a major change
Testing	In enterprise environments, updates are tried out before wide rollout

Table 2.1.17: Update types and maintenance-related concepts.

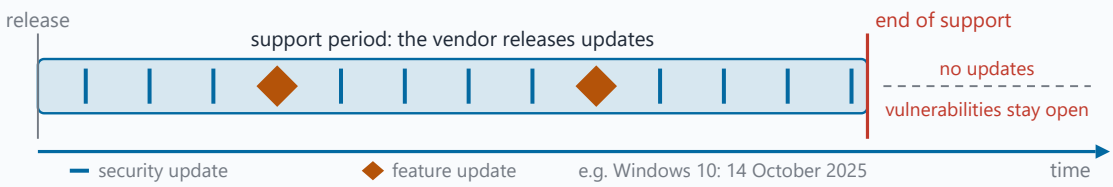


Figure 2.1.32: During the support period regular security updates and occasional feature updates arrive. After end of support no updates arrive and newly found vulnerabilities are not closed. Example: support for

Windows 10 ended on 14 October 2025.

Figure 2.1.32 shows the lifetime of a release. Security updates are frequent and regular; Microsoft releases them on the second Tuesday of every month. Feature updates are less frequent.

End of support does not mean the computer will stop working. A computer with Windows 10 installed still starts after 14 October 2025, but security holes found after that date are not closed. An unsupported system becomes more vulnerable with every passing month. A backup is made before a major update; organizations first try the update on a few test computers, then roll it out in stages.

2.1.12 Virtualization and containers

Two technologies are used to try both Windows and Linux on the same computer, or to share a single powerful server among dozens of customers: virtual machines and containers. Both provide isolation, but they build the isolation at different layers.

2.1.12.1 Virtual machines and hypervisors

Definition 2.1.29

Virtualization is sharing the resources of a single physical computer among several isolated environments. A **virtual machine (VM)** is a virtual computer created in software on a physical computer, with a full operating system running inside it. The system running in a virtual machine is called the **guest operating system**. A **hypervisor** is the software that provides processor, memory, and device resources to virtual machines and isolates them from one another.

The hypervisor does to operating systems, one level up, what the operating system does to processes. Each guest system believes it has its own processor, memory, and disk. In reality the hypervisor divides up the physical resources: on a server with 32 GB of RAM, four virtual machines can be given 8 GB each. If one virtual machine crashes or catches malware, the others are not affected.

2.1.12.2 Containers

Definition 2.1.30

A **container** is a package that isolates an application and its **dependencies**, that is, the libraries and settings it needs to run, at the process level. Containers share the kernel of the host operating system. The software that starts and manages containers is called a **container engine**; the best-known example is Docker.

A container is a solution to the "it worked on my computer" problem. The application is packaged together with the library versions it needs; wherever the package is run, the environment inside it is the same. Because containers have no kernel of their own, they start in seconds and use little memory.

Virtualization and containers are common in four areas: **testing** software on different systems, **consolidating** many lightly used servers onto a single physical machine, **cloud** services, and distributing applications so that they are **portable** to any environment in the same form.

2.1.12.3 Comparing virtual machines and containers

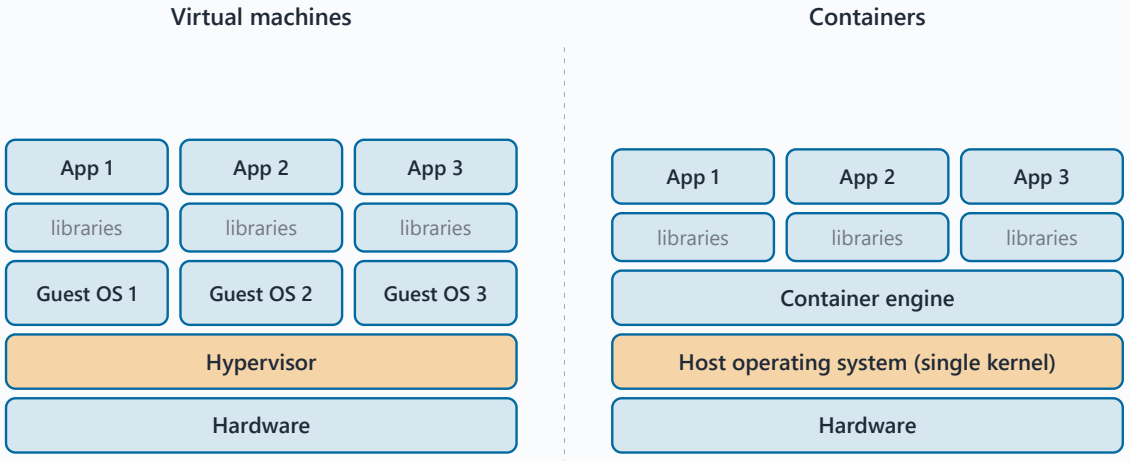


Figure 2.1.33: Virtual machines on the left: above the hypervisor, each application has its own guest operating system and libraries. Containers on the right: the container engine runs on a host operating system with a single kernel; each container carries only its libraries and its application.

Figure 2.1.33 shows the layers of the two approaches side by side. The difference lies in the layer at which isolation is built.

Attribute	Virtual machine	Container
What does it carry?	A full guest operating system and the application	Only the application and its dependencies

Attribute	Virtual machine	Container
Kernel	Each virtual machine has its own kernel	The host operating system's kernel is shared
Isolation	Strong	Weaker
Resource use	More memory, slower start-up	Lightweight, fast start-up
Different operating system	Can run (Windows on top of Linux)	Needs the same kernel as the host

Table 2.1.18: Comparison of a virtual machine and a container.

Example 2.1.23

A server running Linux has two needs: (a) 20 copies of a web application that runs on the same Linux version, (b) an accounting program that runs only on Windows. Which technology fits each need?

1. **(a) Web application:** the application runs on the Linux kernel; it can share the host's kernel. Installing 20 full operating systems for 20 copies would waste memory. Containers fit: each copy is lightweight and starts in seconds.
2. **(b) Accounting program:** the program needs a Windows kernel. Containers share the host's Linux kernel, so they cannot run this program. A virtual machine fits: a full Windows guest system is installed on top of the hypervisor.

2.1.13 Summary

- **Software** is the whole of programs, data, and documentation. System software manages the hardware, application software does the user's work, programming tools produce software, and firmware is embedded in the device. Each layer **abstracts** the one beneath it.
- The **operating system** manages processes, memory, files, devices, and users, and provides network services. **Booting**: power → UEFI → POST → boot loader → kernel → sign-in. **Secure Boot** checks signatures and the **TPM** stores measurements; MBR is limited to 2 TB and 4 primary partitions, GPT is not.
- The **kernel** runs in privileged mode; applications request services from it with a **system call**. Kernels are monolithic, micro, or hybrid.
- A **process** passes through the states New, Ready, Running, Waiting, and Terminated; a process whose wait is over becomes Ready first. **Multiprogramming** runs another process while one waits for I/O. **Round-robin scheduling** gives each process at most one time slice; SJF minimizes the average waiting time.
- A **deadlock** requires all of mutual exclusion, hold and wait, no preemption, and circular wait.

- The base/limit model protects memory; **swapping** frees up room but is slow. In **virtual memory** an address is split into a page and an offset; if the page is not in RAM, a **page fault** occurs. **LRU** evicts the page with the oldest last use, **FIFO** the page that entered earliest.
- **Drivers** translate common requests into hardware operations; devices report with interrupts. The **file system** manages names, metadata, and blocks; an opened file is given a **file descriptor**.
- The command line suits automation; `chmod` writes permissions with the weights r = 4, w = 2, x = 1. **Authentication** answers "who?" and **authorization** answers "what can they do?"
- Personal, mobile, server, embedded, and **real-time** systems have different priorities. The **license** determines the rights; after support ends, no security updates arrive. A **virtual machine** gives strong isolation with its own kernel, a **container** gives lightness with a shared kernel.

Question	Rule	Example
Turnaround time	finish − arrival	P2 in RR: 9 − 0 = 9 ms
Waiting time	turnaround time − processor time	P2 in RR: 9 − 3 = 6 ms
Base/limit	base + address if address < limit	150 < 200 → 100 + 150 = 250
Page and offset	address ÷ 4096 and address mod 4096	4100 → page 1, offset 4
Physical address	frame × 4096 + offset	5 × 4096 + 4 = 20484
Disk blocks	size ÷ block size, rounded up	9 KB, 4 KB block → 3 blocks
chmod digit	r = 4, w = 2, x = 1 added up	rw- → 6; rw-r--r-- → 644
MBR limit	2^{32} sectors × 512 bytes	≈ 2.2 TB

Table 2.1.19: Calculation rules used in these notes.

2.1.14 Key terms

These are the key terms used in this chapter.

software system software application software firmware utility middleware operating system boot boot loader boot order UEFI / BIOS power-on self-test (POST) EFI system partition Secure Boot Trusted Platform Module (TPM) partition / partitioning scheme kernel kernel mode / user mode system call interrupt monolithic / micro / hybrid kernel Linux distribution process process ID (PID) thread process state ready queue scheduling / scheduler context switch concurrency multiprogramming time sharing round robin time slice (quantum) Gantt chart deadlock mutual exclusion hold and wait no preemption circular wait swapping / swap space virtual memory virtual address page / frame page table offset memory management unit (MMU) page fault page file page replacement least recently used (LRU) segmentation fault device driver plug and play buffer file system metadata journaling format root / folder / subfolder path / absolute / relative path

current directory file descriptor handle end of file graphical user interface (GUI) command-line interface (CLI) permission authentication authorization principle of least privilege sandbox real-time system license open source / closed source update virtualization virtual machine hypervisor guest / host operating system container / container engine dependency

2.1.15 Test Yourself

1. Three processes are ready at t = 0; the order is P1, P2, P3; the times are P1 = 3, P2 = 4, P3 = 1 ms. In round-robin scheduling the time slice is 2 ms. What are the Gantt chart, the finish times, and the number of context switches?

   Answer

   0–2 P1 (1 left), 2–4 P2 (2 left), 4–5 P3 (finishes), 5–6 P1 (finishes), 6–8 P2 (finishes). Finish times: P3 = 5, P1 = 6, P2 = 8 ms. There are 4 context switches. Waiting times are P1 = 6 − 3 = 3, P2 = 8 − 4 = 4, P3 = 5 − 1 = 4 ms.

2. The page size is 4,096 bytes; the page table: 0 → frame 2, 1 → frame 5, 2 → on disk, 3 → frame 0. Which page and which offset does virtual address 13000 fall in? What is the physical address?

   Answer

   13000 ÷ 4096 = 3, remainder *13000 − 12288 = 712*: page 3, offset 712. Page 3 is in frame 0: *0 × 4096 + 712 = 712*. Since frame 0 is the very start of RAM, the physical address equals the offset.

3. A process has 3 frames and requests pages in the order 1, 2, 3, 1, 4, 2, 5. How many page faults occur with LRU and with FIFO?

   Answer

   LRU: 1, 2, 3 fault; 1 hit; 4 faults (2, with the oldest last use, is evicted); 2 faults (3 is evicted); 5 faults (1 is evicted) → 6 faults. FIFO: 1, 2, 3 fault; 1 hit; 4 faults (1, the first in, is evicted); 2 hit; 5 faults (2 is evicted) → 5 faults. On this sequence FIFO does better; LRU is not superior in every case.

4. Process C has base register 300 and limit register 250 (KB). C tries to access logical addresses 249

and 250. What does the hardware do for each?

Answer

Since 249 < 250, the access is valid: physical address *300 + 249 = 549*, the last address of C's region. Since 250 ≥ 250, the access is blocked, an interrupt is raised, and the operating system terminates C with an access violation.

5. An operating system requires every process to request all the resources it needs at once, before it starts running. Which of the four deadlock conditions is eliminated? What is the cost of this method?

Answer

Hold and wait is eliminated: a process either gets everything or waits without getting anything; it never waits for another resource while holding one. The cost is that resources are held even during periods when they are not actually used; for example, a process that needs the printer only at the end of its work keeps the printer busy from the start.

6. A 10,000-byte file is written to a disk with a block size of 4,096 bytes. How many blocks does the file occupy and how many bytes are wasted?

Answer

10000 ÷ 4096 ≈ 2.44; rounded up: 3 blocks. It occupies *3 × 4096 = 12288* bytes of space; *12288 − 10000 = 2288* bytes are wasted.

7. (a) What are the permissions after `chmod 700 anahtar.txt`? (b) Which number gives the permission `rwxr-xr-x`?

Answer

(a) 7 = rwx, 0 = ---, 0 = --- → `rwx------`: only the owner can do everything. (b) rwx = 4 + 2 + 1 = 7, r-x = 4 + 1 = 5, r-x = 5 → `755`.

8. 50 copies of a web application running on the same Linux version are needed. Which is more suitable, a virtual machine or a container? Why?

Answer

A container. The application can share the host's Linux kernel; each copy carries only the application and its libraries, starts in seconds, and uses little memory. 50 virtual machines, on the other hand, mean 50 full operating systems and need much more memory and start-up time.

2.1.16 Further reading

- R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (Arpaci-Dusseau Books, 2018; free at ostep.org): covers processes, scheduling, virtual memory, and file systems under the headings of virtualization, concurrency, and persistence, in plain language and with plenty of examples.
- A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts* (10th ed., Wiley, 2018): the standard textbook on operating systems; every topic in these notes, especially scheduling, deadlock, and page replacement, is treated in more detail.
- A. S. Tanenbaum and H. Bos, *Modern Operating Systems* (5th ed., Pearson, 2022): examines the internals of Linux, Windows, and Android through case studies; its chapters on kernel types and virtualization are strong.
- W. Shotts, *The Linux Command Line* (2nd ed., No Starch Press, 2019): teaches the command line, file permissions, and shell scripts from scratch.

2.1.17 References

1. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, Wiley.
2. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, Pearson.
3. The Linux Foundation and Free Software Foundation license resources.
4. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, Arpaci-Dusseau Books.
5. E. G. Coffman, M. Elphick, A. Shoshani, “System Deadlocks”, *ACM Computing Surveys* 3(2), 1971.

PART I · CHAPTER 2.2

The Windows Operating System

The first thing we see when we turn on a computer is the operating system's interface: the icons on the desktop, the taskbar at the bottom, the windows that open, and the folders that hold our files. These notes cover the parts of Windows 11 that people meet most in daily use: the desktop and Start menu, window management, keyboard shortcuts, the organization of files and folders, file paths, File Explorer, the Settings app, accounts and security, storage maintenance, Task Manager, backup, and recovery. Each topic is supported by a mock-up of the screen and a step-by-step table. Numbers such as disk capacity, the space a file takes on disk, a compression ratio, and display scaling are explained with small calculations.

LEARNING OUTCOMES

Interface
Use the Start menu, the taskbar, windows, and virtual desktops; apply commonly used keyboard shortcuts.

File management
Apply the concepts of file, folder, drive, extension, and path; resolve absolute and relative paths step by step; distinguish the consequences of copying and moving.

Settings
Manage account, app, default app, language, and accessibility options.

Maintenance
Choose and use the storage, drive optimization, update, backup, recovery, and security tools that fit the purpose.

2.2.1 Windows and the desktop

An operating system is the basic software that manages the hardware and offers common services to applications. It shares memory among applications, organizes files on the disk, and delivers keyboard and mouse events to the right window. Users do not see most of this work; they deal only with the operating system's interface. In Windows, the center of that interface is the desktop.

2.2.1.1 Windows and its versions

Windows is the operating system that Microsoft develops for personal computers, and it is the most widely used operating system on desktop and laptop machines. Windows uses a window-based graphical interface: each application runs in its own rectangular area, a window, and the user manages the windows with the mouse or the keyboard.

Definition 2.2.1

A **graphical user interface** (GUI) is an interface in which the user interacts with the computer through icons, windows, menus, and buttons. Commands are given not by typing but by clicking items, dragging them, or pressing shortcut keys.

Windows has come in many versions over the years. These notes are written for Windows 11. Support for Windows 10 ended on 14 October 2025. A version that is no longer supported keeps running, but it no longer receives security updates, so a computer connected to the internet should run a supported version.

Windows 11 itself gets one major update a year. Menu names and the positions of some buttons can differ slightly between updates. The names used in these notes are those of the English Windows 11 interface, as in the slides. If a name differs a little on your computer, the option with the same function is usually in the same menu.

Common mistake

Thinking “the computer works, so an old version is not a problem.” Newly discovered security holes are not fixed in a version that is out of support. The computer looks the same every day, but over time it becomes more vulnerable to attacks.

2.2.1.2 The desktop

Definition 2.2.2

The **desktop** is the main work area that appears after sign-in and gives access to applications, files, and system functions. It consists of the wallpaper, the desktop icons, and the taskbar at the bottom of the screen.

Figure 2.2.1 shows the four main parts of the desktop. The **work area** fills most of the screen; the wallpaper shows here and open windows sit on top of it. The **desktop icons** at the top left give quick access to frequently used items: “This PC,” “Recycle Bin,” and the files the user has placed on the

desktop. The **taskbar** stretches along the bottom. At the bottom right is the **notification area** with the clock, network, volume, and battery icons.

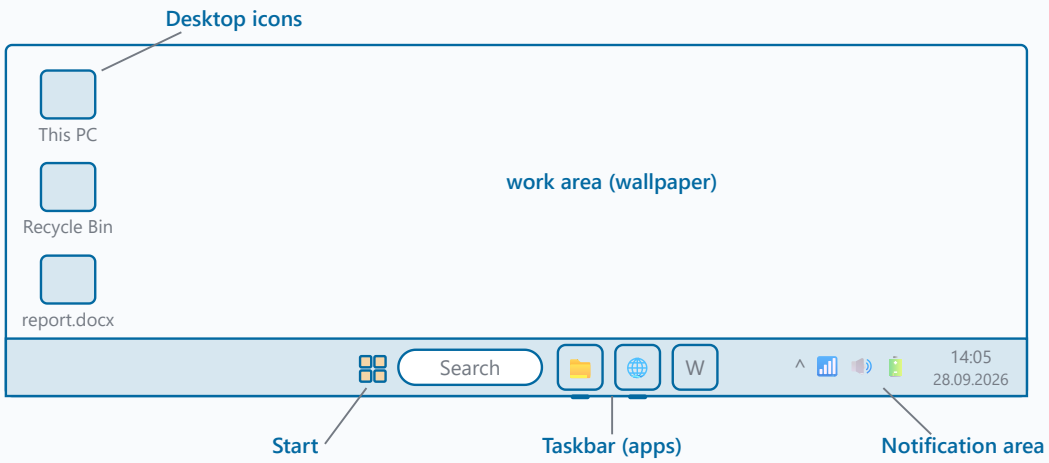


Figure 2.2.1: A mock-up of the Windows desktop. At the top left are the desktop icons (This PC, Recycle Bin, report.docx); at the bottom are the taskbar with the Start button, the search box, and the app icons; at the bottom right is the notification area (network, volume, battery, clock, and date). The rest is the work area, where the wallpaper shows.

The look of the desktop varies from computer to computer. The icons on the taskbar can be centered or left-aligned. The wallpaper, colors, and icon size are changed in the *Personalization* section of the Settings app. So instead of memorizing where things are, it is more useful to know what each item does.

The desktop is also a folder. A file saved to the desktop sits in the user's `Desktop` folder on the disk. Using the desktop as a file store quickly leads to clutter, and a file is hard to find among hundreds of icons. A better habit is to keep on the desktop only the few files and shortcuts being worked on that day and to place everything else in the folder structure described in 2.2.4.2.

2.2.1.3 The Start menu and search

The Windows icon on the taskbar is the **Start button**. Clicking it, or pressing the Windows key (Win) on the keyboard, opens the **Start menu**.

> **Definition 2.2.3**
>
> The **Start menu** is the menu from which one reaches installed applications, recently used files, user account options, and power commands in one place.

Figure 2.2.2 shows the five parts of the Start menu by number:

1. **Search.** This is the box at the top of the menu. It finds apps, files, settings, and help content. Once the Start menu is open there is no need to click the box; typing starts the search.
2. **Pinned.** These are the icons of the apps the user uses often. An app is added here by right-clicking it and choosing "Pin to Start." The *All apps* link at the top right opens an alphabetical list of all installed apps.
3. **Recommended.** Recently opened files and newly installed apps. It is the shortest way back to an unfinished document.
4. **User menu.** It opens when the user name at the bottom left is clicked: account settings, lock, sign out.
5. **Power.** The button at the bottom right: sleep, shut down, restart.

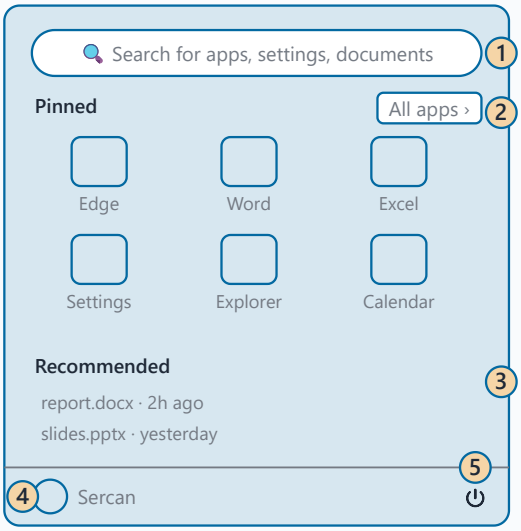


Figure 2.2.2: A mock-up of the Start menu: (1) the search box, (2) the pinned apps and the All apps link, (3) recommended files, (4) the user name, (5) the power button.

Right-clicking the Start button opens another menu. It is called the **Quick Link menu** and can also be opened with the Win + X shortcut. It contains system tools such as Task Manager, Settings, File Explorer, Device Manager, Disk Management, and Terminal. It is the shortest way to a system tool without searching in the Start menu.

Example 2.2.1

You want to change the screen resolution but do not know which menu the setting is in. Search finds it as follows:

1. Press the Win key; the Start menu opens.
2. Type `resolution` right away. There is no need to click the search box.
3. The "Change the resolution of the display" setting appears at the top of the results. When you press Enter, the Settings app opens directly on the System → Display page.

The same method works for apps. Pressing the Win key, typing `word`, and pressing Enter is faster than looking for Word in the menu.

Common mistake

Holding the power button down to shut the computer off. A long press on the case button cuts the power; unsaved documents are lost and the operating system cannot finish half-completed write operations. The computer is shut down with Power → Shut down in the Start menu. Holding the button is a last resort, only when the system responds to no key at all.

2.2.1.4 The taskbar

Definition 2.2.4

The **taskbar** is the bar that is always visible at the bottom of the screen and carries the Start button, the search box, pinned and open apps, the system tray, quick settings, and the clock.

Figure 2.2.3 divides the taskbar into seven parts from left to right. The job of each part is summarized in Table 2.2.1.

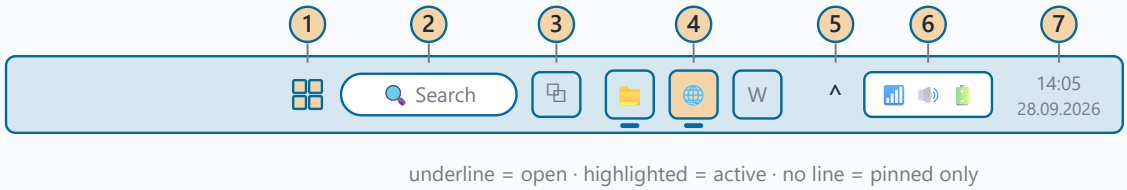


Figure 2.2.3: The parts of the taskbar. (1) Start, (2) search, (3) Task view, (4) app icons, (5) system tray, (6) quick settings, (7) clock and notifications. The line under an app icon shows that the app is open, and a highlighted icon marks the active window; an icon with no line is only pinned.

No.	Part	Job
1	**Start**	Apps, search, account, and power commands
2	**Search**	Find files, apps, and settings
3	**Task view**	Show open windows and virtual desktops
4	**Apps**	Launch pinned apps, switch between open windows
5	**System tray**	Icons of apps running in the background (under the ^ arrow)
6	**Quick settings**	Network, volume, and battery icons; clicking opens the Quick settings panel
7	**Clock and notifications**	Date and time; clicking opens the calendar and notification center

Table 2.2.1: The parts of the taskbar and their jobs.

The fourth part, the app icons, is the most used. An icon can be in one of three states. A **pinned** icon stays on the taskbar even when the app is closed, and clicking it launches the app. An **open** app has a short line under its icon. The icon of the window that currently receives keyboard input is highlighted more strongly. If an app has several windows open, hovering over its icon shows thumbnails of the windows, and you pick the one you want.

To pin an app to the taskbar, right-click its icon while the app is open and choose “Pin to taskbar.” The “Unpin from taskbar” command in the same menu removes the icon; the app itself is not deleted from the computer.

The fifth part, the **system tray**, is easy to overlook at first. Programs such as cloud sync, antivirus, or a music app run in the background without a window. Their icons appear in the small panel that opens when you click the ^ arrow. To close a background app completely, right-click its icon and choose “Exit” or “Quit.”

2.2.1.5 Quick settings and notifications

The two areas at the far right of the taskbar open two small but frequently used panels (Figure 2.2.4).

Definition 2.2.5

Quick settings is the area that gathers frequently changed settings such as Wi-Fi, Bluetooth, airplane mode, volume, brightness, and accessibility in a single panel; it opens by clicking the network, volume, and battery icons or with Win + A. The **notification center** is the panel that shows app notifications and the calendar; it opens by clicking the clock or with Win + N.

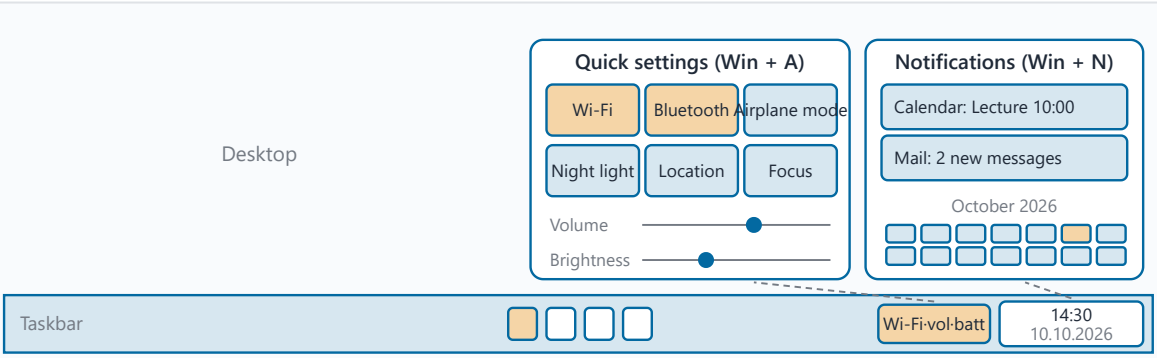


Figure 2.2.4: The two panels that open from the right end of the taskbar. The network, volume, and battery icons open the Quick settings panel (the Wi-Fi, Bluetooth, Airplane mode, Night light, Location, and Focus buttons, and the volume and brightness sliders). The clock opens the Notifications and calendar panel.

The buttons in the Quick settings panel switch on and off with a click. Airplane mode, for example, cuts all wireless connections at once. Night light reduces the blue light of the screen and eases eye strain in the evening. The arrow next to the Wi-Fi button opens the list of nearby wireless networks. The steps for connecting to a network are given in 2.2.8.2.

A notification is a short message an app gives the user: a new e-mail, an upcoming class, a finished download. Notifications appear in a corner of the screen for a few seconds and then pile up in the notification center. Distracting notifications can be turned off per app. For this, switch off the relevant app's toggle on the Settings → System → Notifications page. The **Do not disturb** mode on the same page silences notifications at set times, for example during a class or an exam.

Common mistake

Dismissing all notifications without reading them. Critical system warnings such as "The disk is almost full," "A restart is needed for the update," or "Virus & threat protection is off" also arrive as notifications. These warnings should be read before they are dismissed. Only distracting app notifications should be turned off.

2.2.2 Window management

Windows takes its name from windows. Each application open at the same time runs in its own window. While writing an assignment, you may have Word on one side, a browser with the sources open on the other, and a music app playing in the background. Arranging windows quickly makes it easier to carry on several tasks together.

2.2.2.1 The parts of a window

Definition 2.2.6

A **window** is the rectangular area an application occupies on the screen, one that can be moved and resized. Almost every window has a title bar, control buttons, a work area, and a border.

Figure 2.2.5 shows the parts of a window on a File Explorer window. Although applications differ, most of these parts are in the same place in every window:

- The **title bar** is the strip at the very top. It shows the name of the application and the open document: "Documents - File Explorer." The window is moved by grabbing this bar and dragging; double-clicking the bar maximizes the window or returns it to its previous size.
- The **control buttons** are the three buttons at the right of the title bar: Minimize (—), Maximize (□), and Close (×).
- The **menu or toolbar** carries the application's commands. In File Explorer, the New, Cut, Copy, Paste, and View commands are here; in Word, this area takes the form of a ribbon.
- The **address and search bar** is found in some applications. In File Explorer it shows the path of the folder that is currently open.

- The **work area** is the real content of the window: the file list, the document text, the web page.
- The **scroll bar** appears when the content does not fit in the window and scrolls the content up and down.
- The **status bar** is the thin strip at the bottom. In File Explorer it shows the number of items in the folder and the number of selected items; in Word it shows the page and word counts.
- The **border** is the outer line of the window. When the mouse is over the border or a corner, the cursor turns into a double arrow; dragging resizes the window.

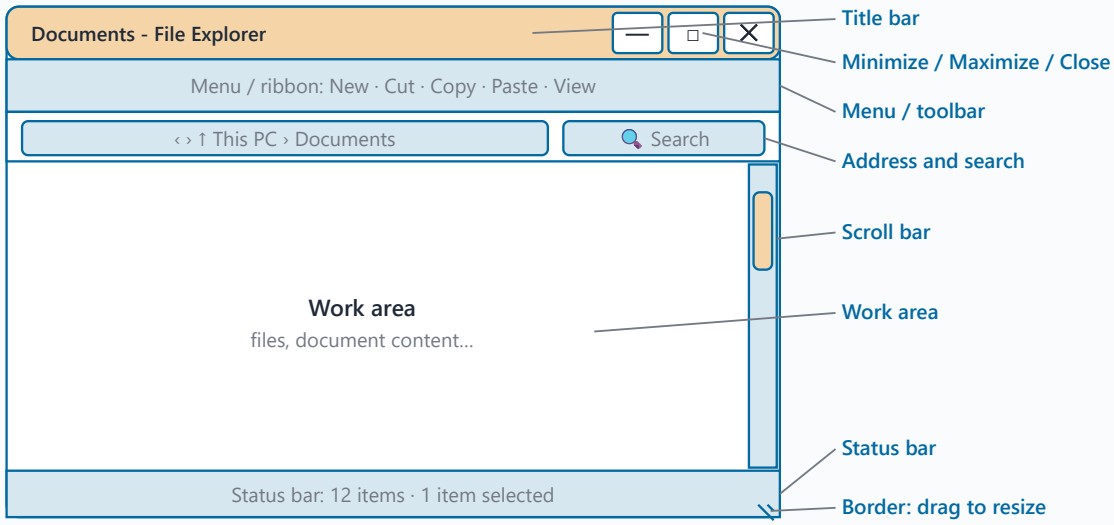


Figure 2.2.5: The parts of a window: the title bar, the Minimize / Maximize / Close buttons, the menu or toolbar, the address and search bar, the work area, the scroll bar, the status bar, and the border.

2.2.2.2 Minimize, maximize, close

A window is in one of three states: **normal** (it covers part of the screen and can be moved and resized), **maximized** (it fills the whole screen), or **minimized** (it is not visible on the screen; only its icon stays on the taskbar). The control buttons move the window between these states:

- **Minimize** (—) sends the window to the taskbar. The application does not close; it keeps running. Clicking its icon on the taskbar brings the window back to where it was.
- **Maximize** (□) spreads the window over the whole screen. Once the window is maximized, the icon of the same button changes and its name becomes **Restore**; pressing it returns the window to its previous size.
- **Close** (×) closes the window. If it is the application's only window, the application closes too. If there are unsaved changes, the application first asks whether to save them.

Common mistake

Confusing minimizing with closing. A minimized window disappears from the screen, but the application keeps running in memory. When dozens of windows are minimized and forgotten, memory fills up and the computer slows down. The lines under the taskbar icons show which applications are still open. A window you are finished with is not minimized; it is closed.

2.2.2.3 Snapping and switching between windows

Resizing windows by hand with the mouse is slow. Windows offers a layout feature that quickly places windows along the edges and corners of the screen.

Definition 2.2.7

Snap is the window-layout feature that fits a window into a half, a third, or a quarter of the screen. A window is snapped by dragging it by the title bar to the edge of the screen, with Win + arrow keys, or by choosing one of the snap layouts that open with Win + Z.

Figure 2.2.6 shows four of the snap layouts. Win + ← places the active window in the left half of the screen and Win + → in the right half. Win + ↑ maximizes the window, while Win + ↓ first restores its previous size and, pressed once more, minimizes it. Win + Z, or hovering the mouse over the Maximize button, shows ready-made layouts such as three columns, four quarters, or one large and two small windows.

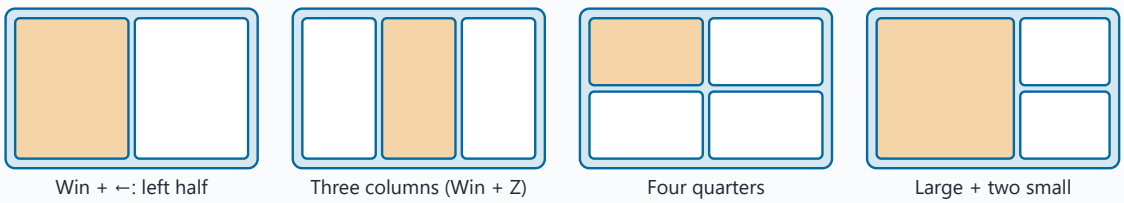


Figure 2.2.6: Snap layouts: the left half with Win + ←, and the three-column, four-quarter, and one-large-two-small layouts that open with Win + Z.

The fastest way to switch between open windows is the **Alt + Tab** shortcut. While the Alt key is held down, pressing Tab brings up thumbnails of the open windows in the middle of the screen. As long as Alt stays down, each Tab selects the next thumbnail; when Alt is released, the selected window comes to the front. Pressing Alt + Tab once and releasing it switches back and forth between two windows.

Let us follow how these operations work together in an example. Figure 2.2.7 shows three key moments of the step-by-step slide in the presentation, and Table 2.2.2 shows all of the steps.

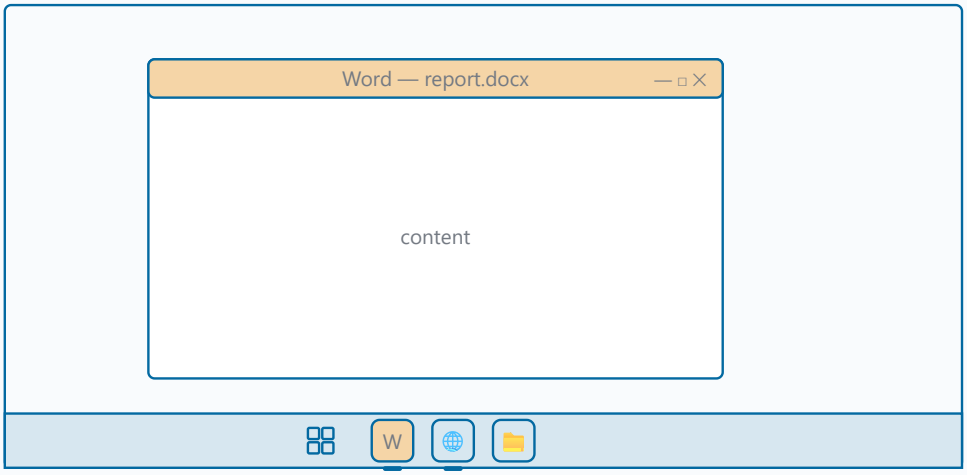
Example 2.2.2

A student is writing a report. `report.docx` is open in Word, and a source article is open in the browser. The student wants to arrange the screen so that they can read the source and write into the report. At the start, Word is at normal size and in front. The operations go as follows:

Step	Operation	Word	Browser	Active window
0	start	normal	in the background	Word
1	Minimize (—) in Word	minimized (line remains)	in the background	—
2	Click the Word icon, then Maximize (□)	maximized	in the background	Word
3	Win + ←	left half	in the background	Word
4	Switch to the browser, then Win + →	left half	right half	Browser
5	Alt + Tab, select Word	left half	right half	Word
6	Close (×) in Word	closed (asks to save if needed)	right half	Browser

Table 2.2.2: Window operations and the state after each one.

In the first step Word does not close; the line under its icon shows that the application is open. In the second step the name of the Maximize button becomes Restore. After the fourth step, two windows sit side by side. In the fifth step Alt + Tab does not change the layout; it only changes which window the keyboard types into. In the last step only the Word window closes, and the browser stays where it is.



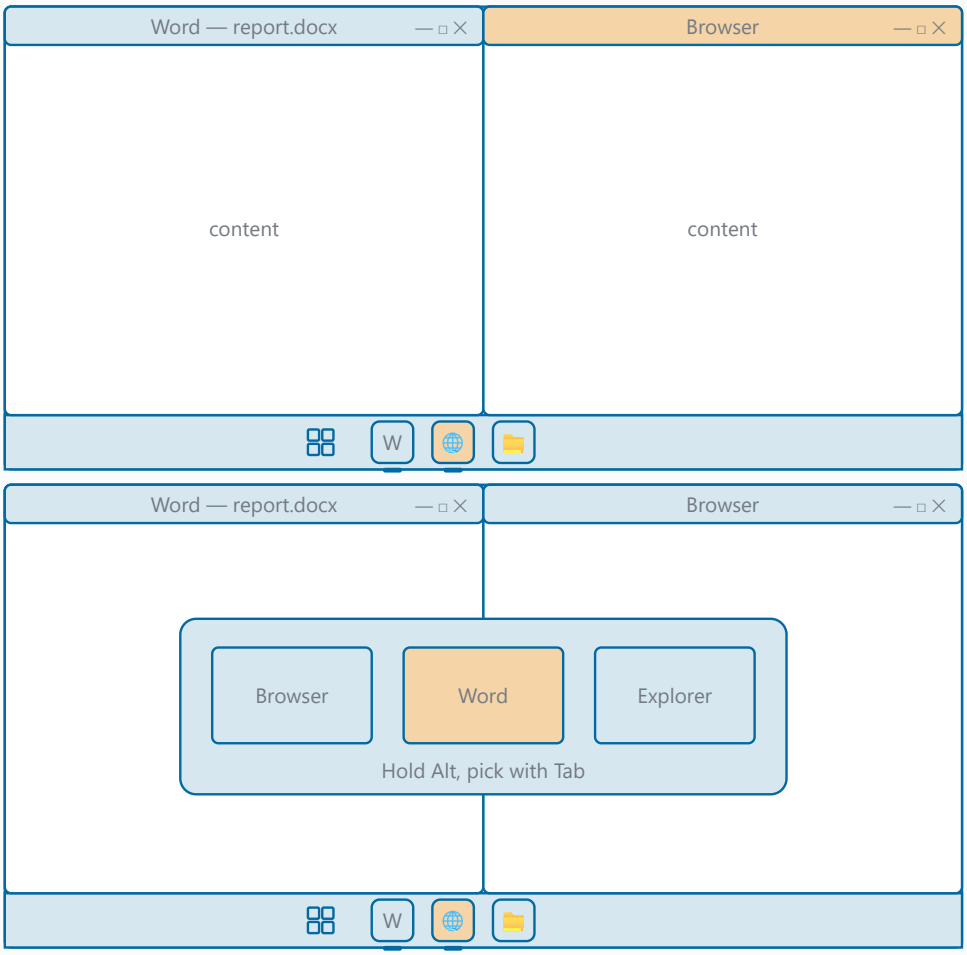


Figure 2.2.7: Three moments from the window states. At the top, Word is at normal size. In the middle, Word is snapped to the left half with Win + ← and the browser to the right half with Win + →. At the bottom, Tab is pressed while Alt is held down, and thumbnails of the open windows appear.

Common mistake

Thinking that Alt + Tab rearranges the windows. Alt + Tab changes only the active window. When you switch between two snapped windows, both stay in place; only the window that comes to the front and receives keyboard input changes.

2.2.2.4 Virtual desktops

When many windows are open at once, the screen gets crowded. The Word and browser windows opened for class get mixed up with music and game windows. Windows solves this problem by dividing windows into separate workspaces.

Definition 2.2.8

A **virtual desktop** is a separate workspace that holds a group of open windows. At any moment the screen shows the windows of only one desktop; the applications on the other desktops keep running in the background.

Virtual desktops are managed with four shortcuts (Table 2.2.3). To see all desktops together, **Task view** is used. Task view opens with the button on the taskbar or with Win + Tab. It shows the open windows of the active desktop at the top and the list of desktops below.

Operation	Shortcut	Result
Task view	Win + Tab	shows the open windows and desktops
New desktop	Win + Ctrl + D	creates an empty desktop and switches to it
Switch	Win + Ctrl + ← / →	switches to the desktop on the left or right
Close	Win + Ctrl + F4	closes the active desktop; its windows move to the desktop on the left

Table 2.2.3: Virtual desktop shortcuts.

Example 2.2.3

Word and a browser are open on Desktop 1 for class. The student wants to take a break to listen to music and play a game, but does not want to close the class windows. The shortcuts are applied in order:

Step	Operation	Desktop 1	Desktop 2	On screen
0	start	Word, Browser	—	Desktop 1
1	Win + Ctrl + D	Word, Browser	(empty)	Desktop 2
2	Music and Game open	Word, Browser	Music, Game	Desktop 2
3	Win + Ctrl + ←	Word, Browser	Music, Game	Desktop 1
4	Win + Ctrl + →	Word, Browser	Music, Game	Desktop 2
5	Win + Ctrl + F4	Word, Browser, Music, Game	— (closed)	Desktop 1

Table 2.2.4: Virtual desktops, step by step.

In the first step the windows on Desktop 1 do not close; they are only withdrawn from the screen. In the third step the class windows return exactly as they were. Meanwhile the music on Desktop 2 keeps playing. In the last step Desktop 2 closes, but the Music and Game windows in it do not close; they move to Desktop 1.

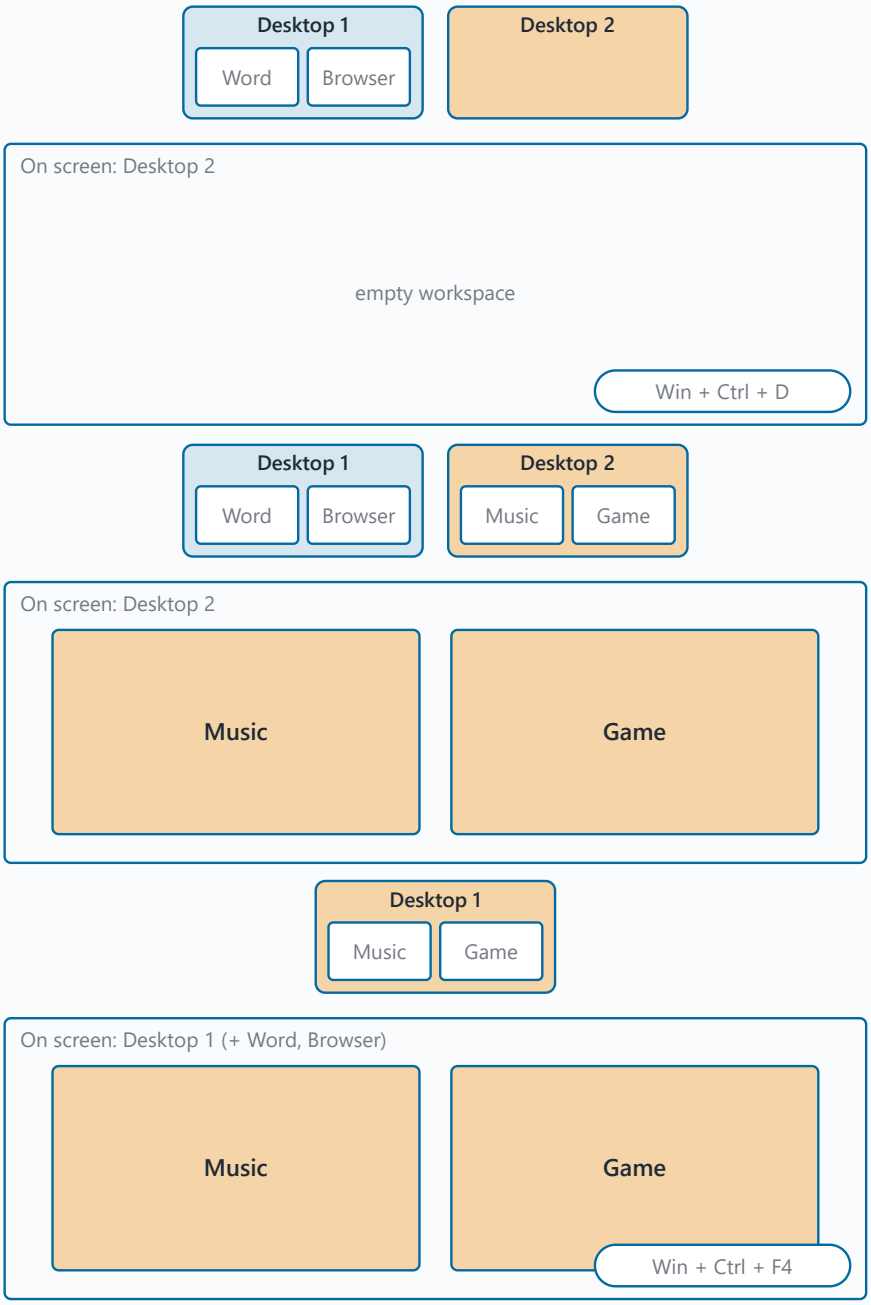


Figure 2.2.8: Three moments from the virtual desktops. At the top, an empty Desktop 2 has been created with Win + Ctrl + D and switched to. In the middle, Music and Game are open on Desktop 2. At the bottom, Desktop 2 has been closed with Win + Ctrl + F4 and its windows have moved to Desktop 1.

Common mistake

Thinking that closing a desktop also closes the applications in it. Win + Ctrl + F4 only removes the workspace; its windows move to the desktop on the left and keep running. To close an application, use the × button on its window. The reverse also holds: switching to another virtual desktop does not stop applications. A game on a desktop that is not on screen still uses

memory and the processor.

2.2.3 Keyboard shortcuts and screenshots

Every move with the mouse means opening a menu, finding an item, and clicking. Doing the same job with a key combination is often several times faster. That is why knowing the shortcuts for frequent tasks saves a great deal of time in daily use.

Definition 2.2.9

A **keyboard shortcut** is a way of running a command by pressing several keys together instead of choosing it from a menu. The notation "Ctrl + C" means pressing the C key while holding the Ctrl key, then releasing both together.

Each of the modifier keys used in shortcuts has roughly its own area. **Ctrl** usually runs commands inside an application (copy, paste, save, undo). The **Win** key reaches Windows' own functions (Settings, File Explorer, window layout). The **Alt** key is used for switching between windows and for reaching menus. **Shift** gives a "stronger" form of a command: Delete sends a file to the Recycle Bin, and Shift + Delete deletes it permanently.

2.2.3.1 Basic shortcuts

Table 2.2.5 gives the basic shortcuts of the presentation in two groups. The first four do the same job in almost every application. The rest open Windows' own tools.

Shortcut	Function	Note
Ctrl + C	**Copy**	the selected item is copied to the clipboard; the source does not change
Ctrl + X	**Cut**	the selected item is put on the clipboard to be moved
Ctrl + V	**Paste**	the item on the clipboard is placed at the cursor or in the open folder
Ctrl + Z	**Undo**	the last operation is undone; it also works in File Explorer
Win + E	**File Explorer**	opens a new File Explorer window
Win + I	**Settings**	opens the Settings app
Win + R	**Run**	a small box that opens a program or folder when its name is typed
Win + L	**Lock screen**	the session stays signed in and the screen locks
Win + Shift + S	**Screenshot of an area**	puts an image of the selected area on the clipboard
Ctrl + Shift + Esc	**Task Manager**	opens Task Manager directly
Win + V	**Clipboard history**	shows a list of the most recently copied items

Shortcut	Function	Note
Win + D	**Show desktop**	hides all windows; pressing it a second time brings them back

Table 2.2.5: Basic keyboard shortcuts.

Behind the copy, cut, and paste commands is the **clipboard**. The clipboard is the memory area where a copied or cut item is held temporarily. Normally the clipboard holds a single item, and each new copy overwrites the previous one. The **clipboard history** that opens with Win + V removes this limit: the most recently copied text and images are listed, and the wanted item is pasted by clicking it. Clipboard history is a feature that has to be turned on the first time it is used.

Locking the session with Win + L and signing out are different things. When you lock, the open applications and documents stay exactly as they are. A lock screen appears, and a PIN or password is requested to return. Signing out, on the other hand, closes all applications. When leaving the computer for a short time, locking is enough.

Example 2.2.4

With Word and a browser open, the keys Win + D, Win + E, Win + Shift + S, and Win + L are pressed in turn. After each step the screen shows the following:

1. **Win + D.** The desktop becomes visible; the Word and browser windows are hidden. The applications do not close; the line under their icons remains.
2. **Win + E.** A File Explorer window opens and appears on top of the desktop.
3. **Win + Shift + S.** The screen darkens and a selection bar appears at the top. An area is selected with the mouse; the image of this area is put on the clipboard.
4. **Win + L.** The lock screen appears. Word, the browser, and File Explorer stay open in the background. When the PIN or password is entered, the screen returns to the state it was in before locking.

In none of the four steps does an application close. The shortcuts only change what is visible.

2.2.3.2 Other useful shortcuts

Table 2.2.6 collects shortcuts that are less well known but often useful. Some of them open the menus and panels described in 2.2.1.3 and 2.2.1.5.

Shortcut	Function
Win + X	Quick Link menu (same as right-clicking Start)
Win + A	Quick settings panel
Win + N	notifications and calendar
Win + Z	snap layouts

Shortcut	Function
Win + . (period)	emoji and symbol panel
Shift + Delete	permanent delete (does not go to the Recycle Bin)
Ctrl + T	new tab in File Explorer
right-click → Compress to → ZIP File	compress the selected items into a ZIP archive

Table 2.2.6: Other useful shortcuts.

Shift + Delete deletes a file without sending it to the Recycle Bin; it cannot be restored from there, and Ctrl + Z does not bring it back either. Because the data on the disk does not vanish at once, special recovery software can sometimes rescue the file, but this should not be relied on. Tabs in File Explorer are available in current Windows 11 versions. As in a browser, they let you keep several folders open in tabs within a single window, so there is no need to open two separate windows when moving files between two folders.

Common mistake

Using Win + D and Win + L for the same purpose. Win + D only hides the windows; anyone who comes to the computer can reopen the documents with one click. When leaving the computer in a place where other people are present, lock the screen with Win + L.

2.2.3.3 Taking a screenshot

To report an error message, to show where a setting is, or to add a chart to a report, you need to capture an image of the screen. Windows has several ways to do this. They differ in two questions: *what* is captured and *where* the image goes (Table 2.2.7).

Definition 2.2.10

A **screenshot** is the current image of the whole screen or part of it, saved as a picture. In Windows, screenshots are taken with the **Snipping Tool** and the PrtSc (Print Screen) key.

Shortcut	What is captured?	Where does the image go?
Win + Shift + S	a selected area: rectangle, freeform, window, or full screen	to the clipboard; it can be opened from the notification to edit and save
Win + PrtSc	full screen	as a file in the Pictures\Screenshots folder (PNG)
Alt + PrtSc	active window only	to the clipboard; paste it into a document with Ctrl + V
PrtSc	opens the Snipping Tool in current Windows 11	if this option is turned off, the full screen is copied to the clipboard

Table 2.2.7: Ways of taking a screenshot.

The most important distinction in the table is whether the image is a *file* or *clipboard* content. Win + PrtSc creates a PNG file; the file stays even when the computer is shut down. The other methods put the image on the clipboard. An image on the clipboard is not a file. It is overwritten at the next copy and is lost when the computer restarts. To keep it, paste it into a document or a program such as Paint and save it, or use the Win + V clipboard history. After an image taken with Win + Shift + S, a notification appears in the corner. Clicking it opens the image in the Snipping Tool, where it can be cropped, drawn on, and saved as a file.

Common mistake

Sharing a screenshot without checking it. The image may contain an e-mail address, a password hint, an open chat notification, or the titles of other tabs. Look at the image before sharing it; if necessary, remove personal information by cropping or painting over it.

2.2.4 Drives, files, and folders

Every document, photo, and program on a computer sits on the disk as a file. The operating system places these files in folders and the folders on drives. This section describes the building blocks of this organization from the bottom up, that is, from the physical disk to the file name.

2.2.4.1 Disks, partitions, and drive letters

In everyday speech the words “disk,” “drive,” and “C:” are used interchangeably. They are in fact three separate concepts.

Definition 2.2.11

A **physical disk** is the device in which data is stored: an SSD, an HDD, or a USB flash drive. A **partition** or **volume** is a logical area set aside on a disk; one disk can be divided into several partitions. A **drive letter** is the name Windows gives to a volume: C:, D:, E:. In File Explorer each letter appears as a separate **drive**.

Figure 2.2.9 shows how these three concepts relate. Physical disk 0 is a 512 GB SSD. This disk is divided into three partitions. The first is a small EFI partition; it is used when the computer starts and has no letter. The second is the 400 GB C: volume, where Windows is installed. The third is the 110 GB D: volume, set aside for data. Physical disk 1 is a 64 GB USB flash drive, and its single partition gets the letter E:. The Z: on the far right is not even a disk: a folder shared by another computer on the network is mapped to this letter.

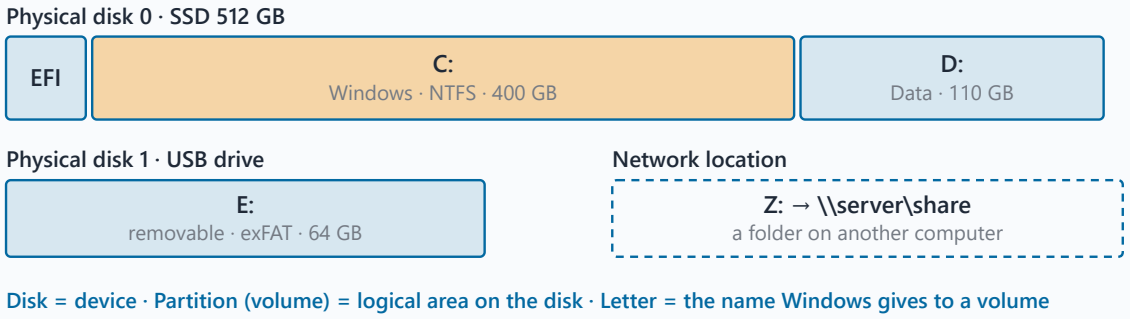


Figure 2.2.9: Two physical disks, their partitions, and drive letters. Disk 0 (SSD, 512 GB) holds the letterless EFI partition, C: (NTFS, 400 GB) where Windows is installed, and the data volume D: (110 GB). Disk 1 (USB drive) is the removable volume E: (exFAT, 64 GB). The letter Z: is mapped to the \\server\share folder on the network.

NTFS and **exFAT** in the figure are names of file systems. A file system is the scheme that keeps the names, sizes, locations, and permissions of the files on a volume. Windows uses NTFS on its own volumes; NTFS supports file permissions, encryption, and large files. USB drives and memory cards usually use exFAT, because Windows, macOS, and many devices can read and write exFAT.

Windows is usually installed on the C: volume, and letters are generally handed out in order. The letters A: and B: were historically reserved for floppy disk drives; that is why the first hard disk volume still gets C: today. When a USB drive is plugged in, it is given the first free letter (E: in this example). For a network share, the user usually picks a letter from the end of the alphabet.

Example 2.2.5

An SSD with “512 GB” on its box is set up as a single partition, C:. Why does Windows show the capacity of this volume as about 476 GB?

1. Disk manufacturers use decimal units: 1 GB = 10^9 bytes. The disk holds 512×10^9 = 512,000,000,000 bytes.
2. Windows, however, calculates with binary units: 1 GiB = 2^{30} = 1,073,741,824 bytes. Even though the screen says “GB,” what is meant is GiB.
3. 512,000,000,000 ÷ 1,073,741,824 ≈ 476.8. Windows shows this number as “476 GB.”

No byte has been lost from the disk; only the unit is different. The EFI and recovery partitions also take up anywhere from a few hundred MB to a few GB, so the C: volume may look a little smaller still.

Common mistake

Thinking that C: and D: are separate disks. They can also be two partitions of the same physical disk. In that case copying files from C: to D: is not a backup: if the disk fails, both volumes are lost together. A real backup is kept on a separate device or in the cloud.

2.2.4.2 Files and folders

Definition 2.2.12

A **file** is a unit of data with a name, contents, and properties; for example, `report.docx`. A **folder** is a logical container that holds files and other folders together; for example, `Documents\Courses`. A folder inside a folder is called a **subfolder**.

Because folders can be nested, files are organized in a tree. Figure 2.2.10 gives a small example. Inside the `Documents` folder there is a `Courses` subfolder, and inside it three files.

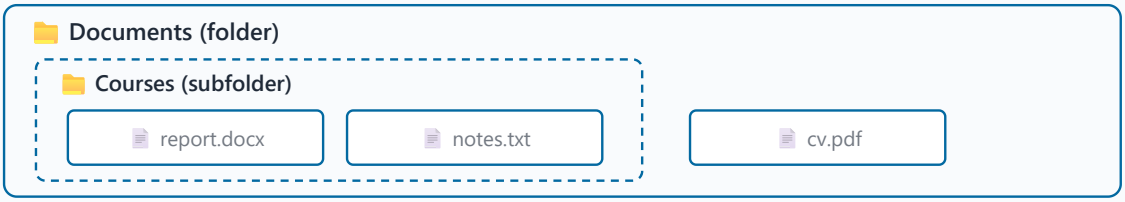


Figure 2.2.10: Inside the Documents folder is the Courses subfolder, and inside it the files report.docx, notes.txt, and cv.pdf.

A folder itself holds no document content. A folder is like a list that knows the names of the files inside it and where they are on the disk. It resembles the shelves in a library: a shelf is not a book, but it makes books findable. If the folder organization is set up well, a file you are looking for among hundreds can be found in a few clicks.

2.2.4.3 File names and extensions

The full name of a file has two parts: in `project_report.pdf`, the part up to the last dot is the **file name** and the part after the last dot is the **extension**.

Definition 2.2.13

An **extension** (file name extension) is the part of a file name after the last dot, and it gives a hint about the file's format: `.pdf`, `.docx`, `.jpg`. When a file is double-clicked, Windows decides which application to open by looking at the extension. This link between an extension and an application is called a **file association**.

The user gives the file name to describe the content. The extension is usually added by the program that creates the file: Word adds `.docx`, Excel `.xlsx`, a camera `.jpg`. Windows hides extensions by default. In File Explorer, extensions become visible when the View → Show → File name extensions option is checked (2.2.6.2).

Figure 2.2.11 shows the two faces of the extension. At the top, each extension leads to its associated application: `.pdf` to the PDF reader, `.xlsx` to Excel, `.jpg` to the Photos app. At the bottom, the name of a PDF file has been changed to `report.docx`. Windows now tries to open this file with Word, but because the content is still a PDF, Word cannot recognize the file.



Figure 2.2.11: The extension determines which application opens a file: project_report.pdf → PDF reader, budget.xlsx → Excel, holiday.jpg → Photos. Renaming report.pdf to report.docx does not convert the content; the content is still a PDF, and Word cannot open the file.

Example 2.2.6

With extensions visible, the file `photo.jpg` is renamed to `photo.png` with F2. What happens?

1. Windows shows a warning that the file may become unusable if the extension changes. If Yes is chosen, the name changes.
2. The bytes inside the file do not change. The content is still a picture compressed in the JPEG format, and its size stays the same.
3. Windows now thinks the file is a PNG. Most picture programs look at the first bytes of the content and recognize the format anyway, but some programs and websites may reject the file.

A real conversion is done in a program that opens the file, with the “Save As” or “Export” command. For example, the picture is opened in Paint and saved as a PNG. This operation re-encodes the content and produces a new file.

Common mistake

Renaming a file while extensions are hidden and adding an extension at the end. A file `homework.docx` with a hidden extension appears on screen as `homework`. If the user renames it `homework.pdf`, the file’s real name becomes `homework.pdf.docx`. The file is still a Word document and is rejected when uploaded to a system that expects a PDF. Malicious files use a similar trick: a program named `invoice.pdf.exe` looks like `invoice.pdf` when extensions are hidden. Keeping extensions visible is therefore a safe habit.

2.2.4.4 File naming rules

Windows places some limits on file names. Some of these limits are mandatory; Windows does not accept a name that breaks the rule. Others are merely good practice.

Forbidden characters. Nine characters cannot be used in a file name: `\ / : * ? " < > |`. Most of them have a special meaning in paths and searches. `\` separates folders, `:` follows the drive letter, and `*` and `?` are wildcards in searches. When you try to type one of these characters, Windows shows a warning and does not accept it.

Reserved names. `CON`, `PRN`, `AUX`, `NUL`, `COM1` … `COM9`, and `LPT1` … `LPT9` are old device names. These names cannot be file names even with an extension; `con.txt` is rejected as well.

Upper and lower case. Windows treats the names `Report.docx` and `report.docx` as the same name, so the two cannot be in the same folder. The capital letter is preserved when the name is typed, but the comparison ignores case.

Path length. The classic limit is 260 characters for the full path, from the drive letter to the end of the file name. In very deep folder structures or with very long names, copying and opening operations can fail because of this limit.

Example 2.2.7

Six file names are tested against the rules:

Name	Valid?	Why
`report:final.docx`	no	`:` is a forbidden character
`question?.txt`	no	`?` is a forbidden character

Name	Valid?	Why
`CON.txt`	no	CON is a reserved name; the extension does not change that
`console.txt`	yes	the name starts with CON but is not CON
`Report.docx` (when `report.docx` is in the folder)	no	upper and lower case do not distinguish names; the name already exists
`homework 1 Ayşe .pdf`	yes	follows the rules, but spaces and non-English characters can cause trouble

The name in the last row is valid in Windows. However, in an e-mail attachment, in an upload to a website, or on another operating system, spaces and letters like `ş` may get garbled. A space before the extension can also cause the name to be misread.

Beyond the mandatory rules, a good file name describes the content and the version. Table 2.2.8 compares two weak names with better versions.

Weak name	Better name	Why?
`final FINAL new (2).docx`	`2026-10-17_report_v2.docx`	date first: sorting by name becomes sorting by date
`homework 1 Ayşe .pdf`	`hw1_name-surname.pdf`	spaces and non-English characters can cause trouble in e-mail and web systems

Table 2.2.8: Weak and better file names.

The date format in the first row is especially useful. If the date is written in year-month-day order with a two-digit month and day, then sorting the names alphabetically also puts them in date order. The names `2026-09-30`, `2026-10-02`, and `2026-10-17` are also alphabetical in that order. The day-month-year format (`17.10.2026`) lacks this property: the name `02.10.2026` comes before `30.09.2026`.

Common mistake

Telling versions apart with names like “final,” “final version,” and “real final.” A few days later, it is unclear which is the newest. Using a version number (`_v1`, `_v2`) or a date keeps versions ordered and clear.

2.2.4.5 Common file types

Recognizing the extension lets you tell what a file is without opening it. Table 2.2.9 lists the most common types by category.

Category	Extensions	Typical use
Document	.docx, .pdf, .txt	text and reports
Spreadsheet	.xlsx, .csv	calculations and data
Presentation	.pptx	slide shows
Image	.jpg, .png, .gif	photos and graphics
Audio / video	.mp3, .wav, .mp4	media
Archive	.zip, .7z	compressed files
Executable	.exe, .msi	program or installer

Table 2.2.9: Common file types.

The differences between some types lie in details that are not visible at first glance. `.txt` holds only plain text; it contains no fonts, bold text, or pictures. `.docx`, on the other hand, also stores formatting, pictures, and tables. `.csv` is plain text made of values separated by commas or semicolons; Excel opens it as a table but it does not store formulas or formatting. `.pdf` freezes a page so that it looks the same on every computer, which makes it suitable for assignments to be handed in.

There is an important distinction among image types, too. `.jpg` (JPEG) is designed for photographs and uses lossy compression: the file gets smaller, but details the eye will hardly notice are discarded. `.png` is lossless; it is better suited for screenshots, drawings, and graphics with sharp edges.

The executable files in the last row need special care. When an `.exe` or `.msi` file is double-clicked, it runs a program on the computer. One might think a document cannot do harm, but an executable file can reach everything on the computer. Executable files and e-mail attachments from unknown sources should not be opened (2.2.9.3).

2.2.5 File paths

Because folders are nested, it is not possible to state a file's location with a single word. Dozens of files named "report.pdf" can sit in different folders. To specify a file unambiguously, we give the sequence of folders that leads from the drive down to the file. This sequence is called the file's path.

2.2.5.1 The parts of a path

Definition 2.2.14

A **path** is the sequence of folders passed through to reach a file or folder, separated by backslashes (`\`). An **absolute path** starts at the root folder of the drive and points to the same file wherever you are. A **relative path** is written relative to the folder you are currently in, the **current folder**.

An example absolute path is:

> `C:\Users\Sercan\Documents\Courses\report.pdf`

Figure 2.2.12 breaks this path into its parts. `C:\` is the **root** folder of the drive. Then come the `Users`, `Sercan`, `Documents`, and `Courses` folders in turn. The last item, `report.pdf`, is the file itself. Each `\` shows a step down one level.

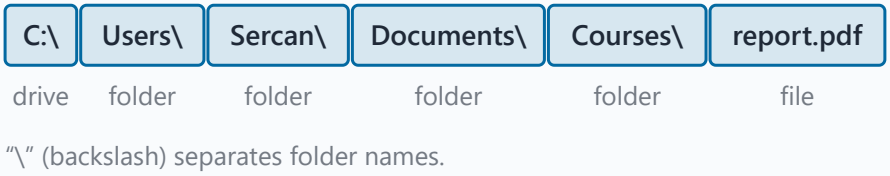


Figure 2.2.12: The parts of a file path: the drive (C:\), four folders (Users, Sercan, Documents, Courses), and the file (report.pdf). The backslash separates folder names.

One detail in this path deserves attention. On a Windows with a non-English interface language, File Explorer may show these folders under translated names (on Turkish Windows, for example, "Belgeler" for Documents), but their real names on disk are English: `Documents`, `Desktop`, `Downloads`, `Pictures`, `Music`, and `Users`. Windows translates the names of these folders into the interface language for display, but real names are used when a path is written. So `C:\Users\Sercan\Documents` and the translated "Belgeler" in Explorer are the same folder.

The length of a path can be counted, too. The path in this example has 44 characters and contains 5 backslashes. That is far below the classic 260-character limit. But as each folder name grows longer and the number of nested folders increases, the limit gets closer. For example, if a course project's folder is ten levels deep under `C:\Users\Sercan\Documents\…` and each level's name is 25 characters long, the path alone exceeds 250 characters.

2.2.5.2 Resolving an absolute path

Resolving a path means finding the item the path points to. An absolute path is read from left to right, and each part goes down one level. Figure 2.2.13 shows this reading on the folder tree. Table 2.2.10 lists the part read at each step and the location reached.

Example 2.2.8

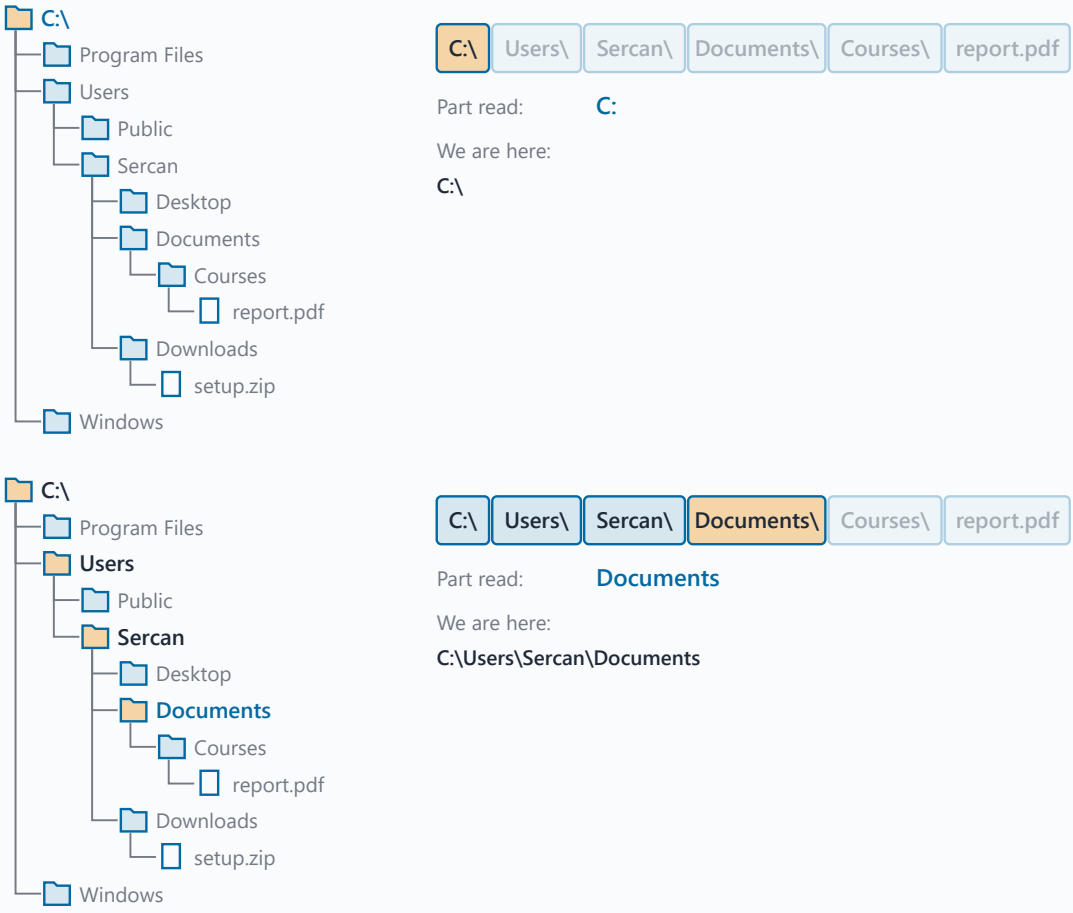
The path `C:\Users\Sercan\Documents\Courses\report.pdf` is resolved step by step.

Step	Part read	Location	Explanation
1	`C:`	`C:\`	the root folder of the drive; the search starts here
2	`Users`	`C:\Users`	Program Files and Windows in the root are skipped

Step	Part read	Location	Explanation
3	`Sercan`	`C:\Users\Sercan`	the user's own folder
4	`Documents`	`C:\Users\Sercan\Documents`	the user's documents folder
5	`Courses`	`…\Documents\Courses`	go down into the subfolder
6	`report.pdf`	`…\Courses\report.pdf`	the last part is a file; found

Table 2.2.10: Resolving an absolute path.

At each step only the inside of the current folder is looked at. At the second step, the root holds the `Program Files`, `Users`, and `Windows` folders; because the path asks for `Users`, the others are never opened. A path works like a map: there is no chance of turning into the wrong folder.



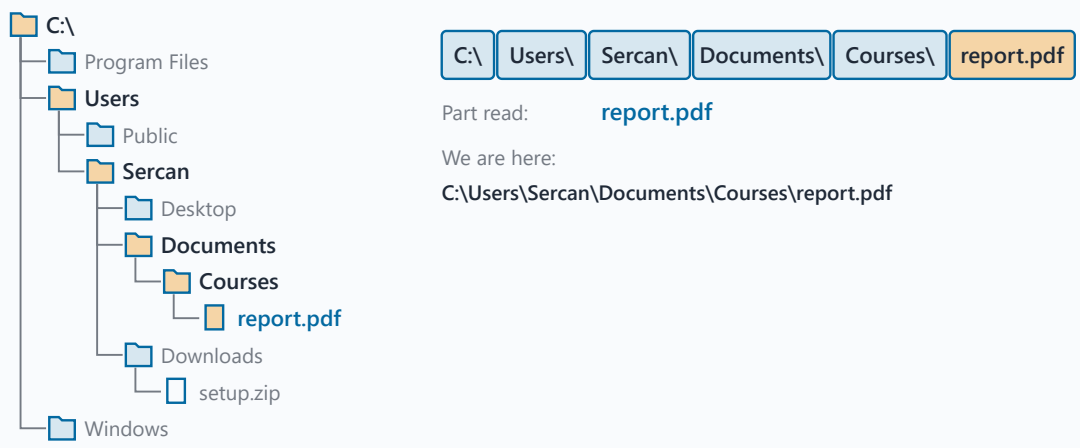


Figure 2.2.13: Resolving an absolute path in the tree. At the top the reading starts at the root C:\. In the middle it passes through Users and Sercan and goes down to Documents. At the bottom it goes down to Courses, and the last part, the file report.pdf, is found.

2.2.5.3 Resolving a relative path

A relative path starts not at the root but at the current folder. It can contain two special names: `..` (two dots) means the **parent folder**, that is, going up one level. `.` (one dot) is the current folder itself. Relative paths are used often in programming and on the command line. When a project folder is moved to another computer, absolute paths break, but relative paths inside the project keep working.

Example 2.2.9

The current folder is `C:\Users\Sercan\Documents`. The relative path `..\Downloads\setup.zip` is resolved.

Step	Part read	Current location
0	—	`C:\Users\Sercan\Documents`
1	`..`	`C:\Users\Sercan` (went up to the parent folder)
2	`Downloads`	`C:\Users\Sercan\Downloads`
3	`setup.zip`	`C:\Users\Sercan\Downloads\setup.zip` (file found)

Table 2.2.11: Resolving a relative path.

The absolute equivalent of the relative path is the path in the last row, `C:\Users\Sercan\Downloads\setup.zip`. It can also be found this way: for each `..`, delete the last part of the current folder's path, then append the remaining parts in order.

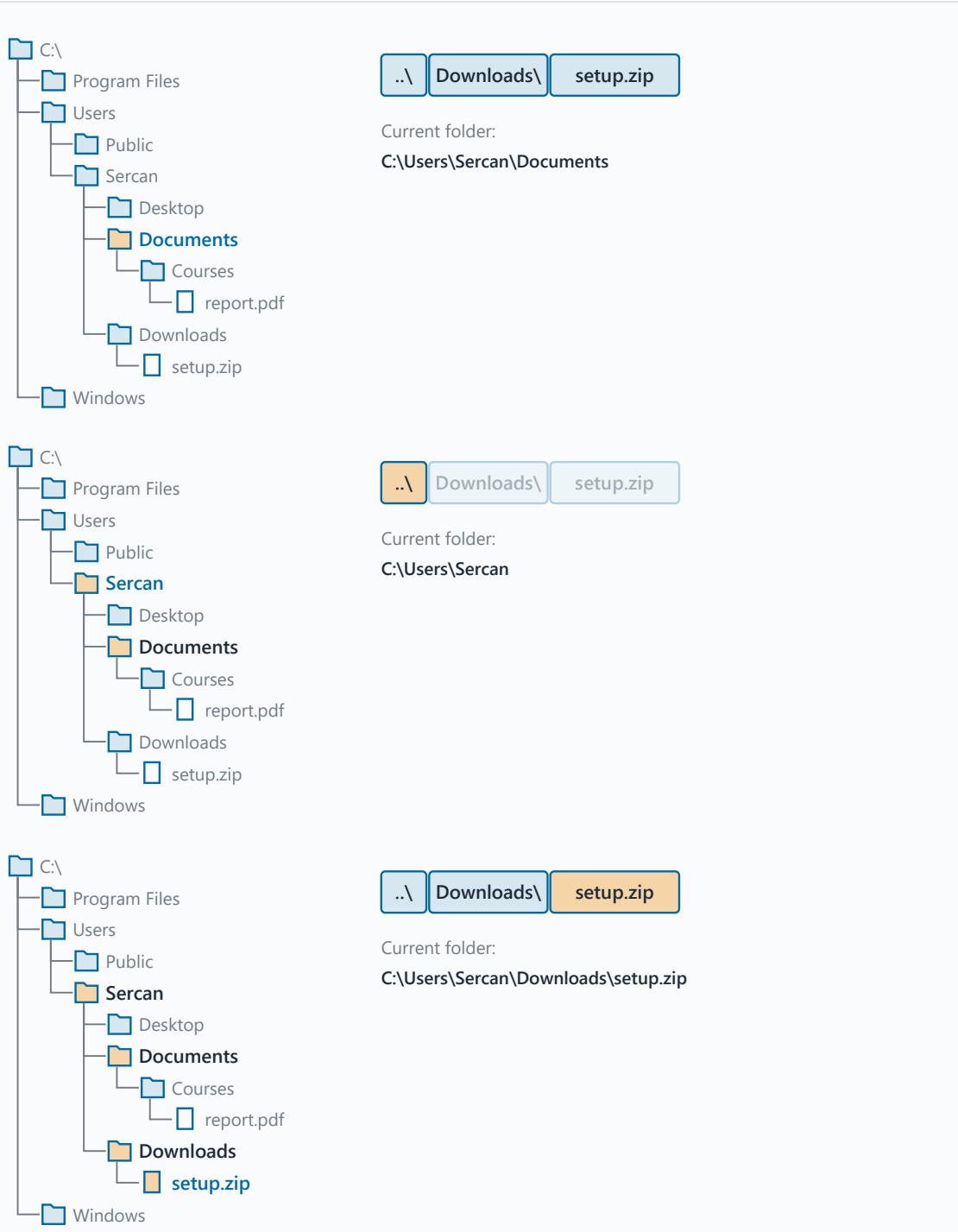


Figure 2.2.14: Resolving a relative path in the tree. At the top the current folder is Documents. In the middle, .. goes up to Sercan. At the bottom it goes down into Downloads and the file setup.zip is found.

Several `..` can follow one another. Each goes up one more level. Let us see this on a longer path.

Example 2.2.10

The current folder is `C:\Users\Sercan\Downloads`. What is the absolute equivalent of the path `..\Documents\Courses\report.pdf`? If the current folder were `C:\Users\Sercan\Documents\Courses`, what would `..\..\Pictures\Holiday\sea.jpg` point to?

1. First path: `..` goes up from `Downloads` to `C:\Users\Sercan`. Then `Documents`, `Courses`, and `report.pdf` are appended. The result is `C:\Users\Sercan\Documents\Courses\report.pdf`, which is exactly the absolute path in 2.2.5.2.
2. Second path: the first `..` goes up from `Courses` to `Documents`, and the second `..` goes up from `Documents` to `Sercan`.
3. Then it goes down into the `Pictures` and `Holiday` folders. The result is `C:\Users\Sercan\Pictures\Holiday\sea.jpg`.

The same file is reached from different current folders with different relative paths. The absolute path is always the same.

Common mistake

Forgetting the `..\` at the start of a relative path. If the current folder is `Documents` and you write `Downloads\setup.zip`, the `Downloads` folder is looked for *inside* `Documents`. Because there is no such subfolder, the file cannot be found. Before writing a relative path, you must know exactly which folder is current.

2.2.6 File Explorer

The basic tool for moving around the folder tree and for viewing and organizing files is File Explorer. It opens with the folder icon on the taskbar or with Win + E.

2.2.6.1 The panes of File Explorer

Definition 2.2.15

File Explorer is the Windows application used to view, open, copy, move, rename, delete, and search for drives, folders, and files.

Figure 2.2.15 divides the File Explorer window into six parts:

1. **Command bar**: the New (folder or file), Cut, Copy, Paste, Rename, Sort, and View commands.
2. **Address bar**: the path of the current folder. The path is shown as clickable parts: *This PC › Documents › Courses*. Clicking a part goes back to that folder. When you click an empty spot on the bar, the full path appears as text; typing a path there and pressing Enter goes straight to that folder.
3. **Search box**: searches the current folder and all of its subfolders.
4. **Navigation pane**: the column on the left. It goes with a single click to frequently used locations such as Home, Desktop, Documents, Downloads, This PC, and Network.
5. **File list**: the contents of the current folder. In the *Details* view, each row shows the name, date modified, type, and size. Clicking a column header sorts the list by that column; clicking once more reverses the order.
6. **Preview / details pane**: the optional column on the right. It shows the contents of the selected file without opening it, or lists its properties. It is turned on and off from the View menu.

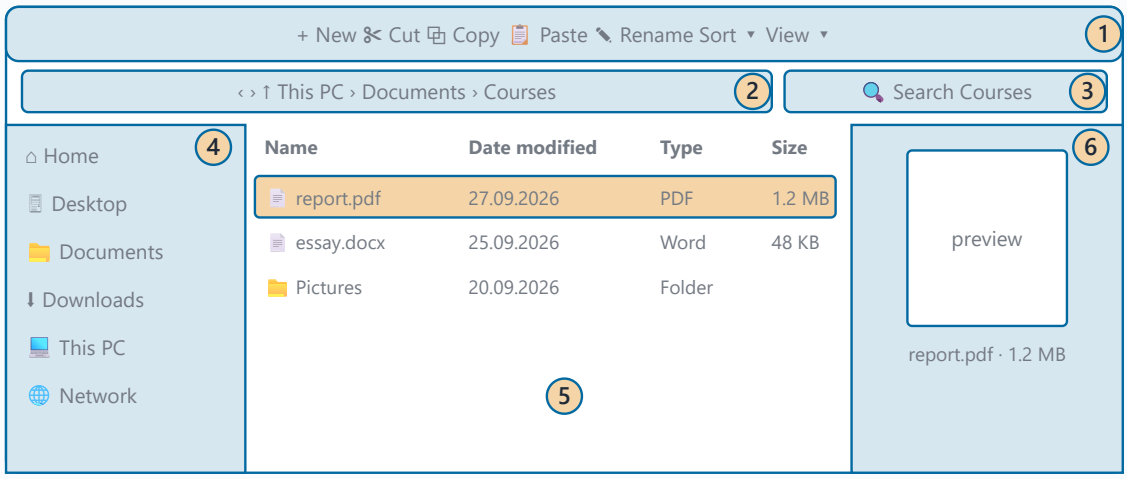


Figure 2.2.15: The panes of File Explorer: (1) the command bar, (2) the address bar, (3) search, (4) the navigation pane, (5) the file list in Details view, (6) the preview of the selected file report.pdf.

The three arrows to the left of the address bar make navigation easier. ← (Back) and → (Forward) move back and forth between visited folders, as in a browser. ↑ (Up) goes up to the parent folder of the current folder; that is, it is the Explorer equivalent of `..` in a relative path. When *This PC* is selected in the navigation pane, the file list shows the drives and a usage bar for each.

2.2.6.2 Extensions, hidden items, new folders, and search

Let us follow five of the most common tasks in File Explorer on an example. At the start, the `Documents\Courses` folder holds four files: `photo.jpg` (2,410 KB), `hw2.pdf` (820 KB), a Word document named `report` (480 KB), and a PDF document named `report` (610 KB). There is also a `draft.txt` (2 KB) with the hidden attribute, but it does not appear in the list. Table 2.2.12 shows the result of each step, and Figure 2.2.16 shows three key moments.

Example 2.2.11

Step	Operation	Items shown in the list
1	View → Show → *File name extensions* is checked	photo.jpg, hw2.pdf, report.docx, report.pdf
2	View → Show → *Hidden items* is checked	the previous four files + draft.txt with a faded icon
3	New → Folder; `Homework` is typed as the name and Enter is pressed	Homework (at the top), then five files
4	report.docx is dragged and dropped onto the Homework folder	Homework, photo.jpg, hw2.pdf, report.pdf, draft.txt
5	`*.pdf` is typed in the search box	hw2.pdf, report.pdf

Table 2.2.12: Five operations in File Explorer and their results.

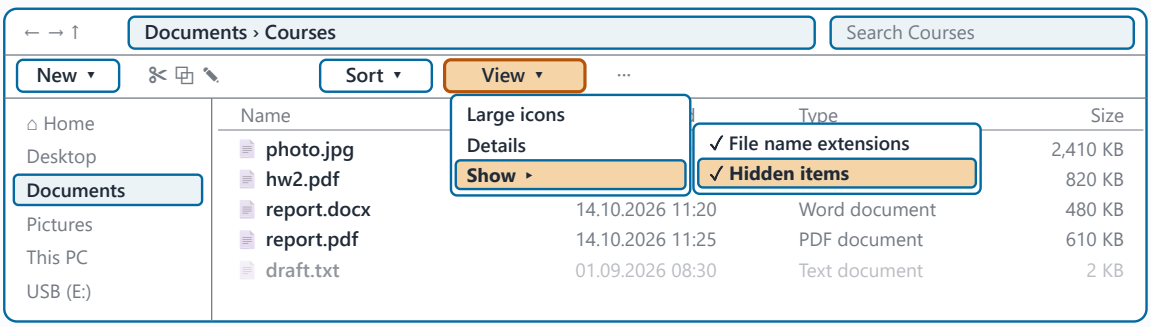
Step 1. While extensions were hidden, both files looked like “report.” Only their icons and the Type column (Word document, PDF document) told them apart. When extensions are turned on, the names separate into `report.docx` and `report.pdf`.

Step 2. When hidden items are turned on, `draft.txt` joins the list with a faded icon. The faded icon indicates that the item carries the hidden attribute.

Step 3. The new folder is created with the name “New folder” and its name comes up selected. `Homework` is typed directly and Enter is pressed; there is no need to press F2 as well. Because folders are listed before files when sorted by name, Homework sits at the top.

Step 4. Because `report.docx` is dragged to a folder on the same drive, it is moved. The file disappears from the Courses list and goes into Homework.

Step 5. `*` is a **wildcard** and means “any name.” The search `*.pdf` finds all files in Courses and its subfolders whose names end in `.pdf`. When the search box is cleared, the full list comes back.



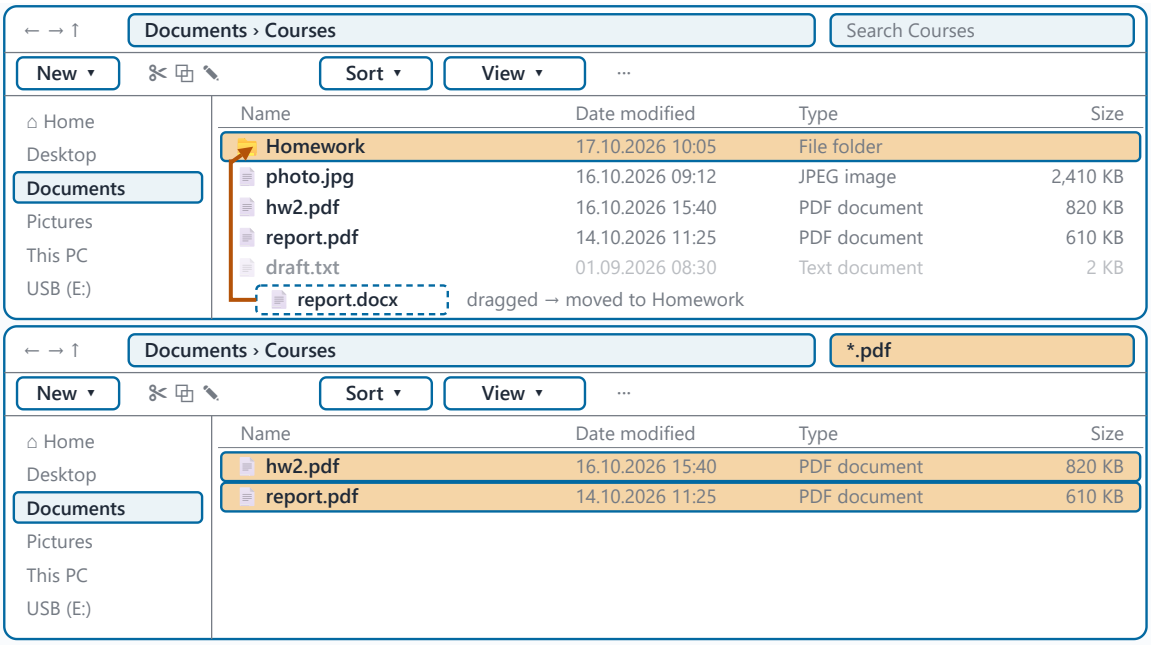


Figure 2.2.16: Three moments in File Explorer. At the top, File name extensions and Hidden items are checked; the hidden draft.txt shows in the list. In the middle, the Homework folder has been created and report.docx has been moved into it. At the bottom, *.pdf has been typed in the search box and only the two PDF files are listed.

The wildcard is used in other forms, too. In the same folder, the search `report.*` finds all files named "report" regardless of type: `report.docx` and `report.pdf`. `hw*` finds everything whose name starts with "hw": in this example, `hw2.pdf`. It is also possible to type a word without a wildcard in the search box. In that case Windows brings up the files that have that word in their name or (in indexed locations) in their content.

Common mistake

Thinking that the hidden attribute protects a file. The hidden attribute only hides the file from the list. Anyone who turns on the "Hidden items" option sees and can open the file. To protect a file from others, use separate user accounts, file permissions, or encryption (2.2.8.3, 2.2.10.5).

2.2.7 File operations

After files are created, they change places, their names are corrected, unneeded ones are deleted, and those to be shared are gathered in an archive. Each of these operations looks simple, but their consequences differ. Was a copy of the file made, or did the file change place? Can a deleted file be brought back? This section answers these questions step by step.

2.2.7.1 Copying and moving

Definition 2.2.16

Copying creates a second, independent instance of an item with the same content at the destination; the original stays at the source. **Moving**, on the other hand, changes the item's location; the item leaves the source and goes to the destination. Copying is done with Copy + Paste (Ctrl + C, Ctrl + V), and moving with Cut + Paste (Ctrl + X, Ctrl + V).

Operation	What happens at the source?	What happens at the destination?
Copy + Paste	the original stays	a new copy appears
Cut + Paste	the item is removed from the source location	the item moves to the destination
Drag and drop	moves within the same drive, copies to a different drive	check which one by looking at the tooltip next to the cursor

Table 2.2.13: The results of copying and moving at the source and the destination.

After copying there are two separate files. When one is changed, the other is not affected. After moving, the number of files does not change; only the file's location does. The Cut command does not remove the file from its place right away. The icon of the cut file fades and the file is marked to be moved. The actual move happens when the Paste command is given. If something else is copied before pasting, the cut operation is canceled and the file stays where it was.

Example 2.2.12

`C:\Documents` holds `report.docx` and `photo.jpg`; the USB drive (E:\) is empty. First the report is copied to the USB drive, then the photo is moved to it. Table 2.2.14 shows the two locations and the clipboard after each step.

Step	Operation	C:\Documents	E:\	Clipboard	Total files
0	start	report.docx, photo.jpg	(empty)	(empty)	2
1	Select report.docx, Ctrl + C	report.docx, photo.jpg	(empty)	report.docx (to be copied)	2
2	Open E:\, Ctrl + V	report.docx, photo.jpg	report.docx	report.docx (to be copied)	3
3	Select photo.jpg, Ctrl + X	report.docx, photo.jpg (faded)	report.docx	photo.jpg (to be moved)	3
4	Ctrl + V in E:\	report.docx	report.docx, photo.jpg	(empty)	3

Table 2.2.14: Copying and moving, step by step.

In the first step nothing changes at the source; the clipboard only remembers which file is to be copied. In the second step the total number of files goes from 2 to 3, because a new copy has been created. In the third step `photo.jpg` is still in place; it only looks faded. In the fourth step the photo goes to the destination and leaves the source. The total stays at 3: moving does not change the number of files. After copying, the clipboard keeps its content, so the same file can be pasted in other places too. A cut item, on the other hand, is moved with a single paste, so the clipboard is emptied.

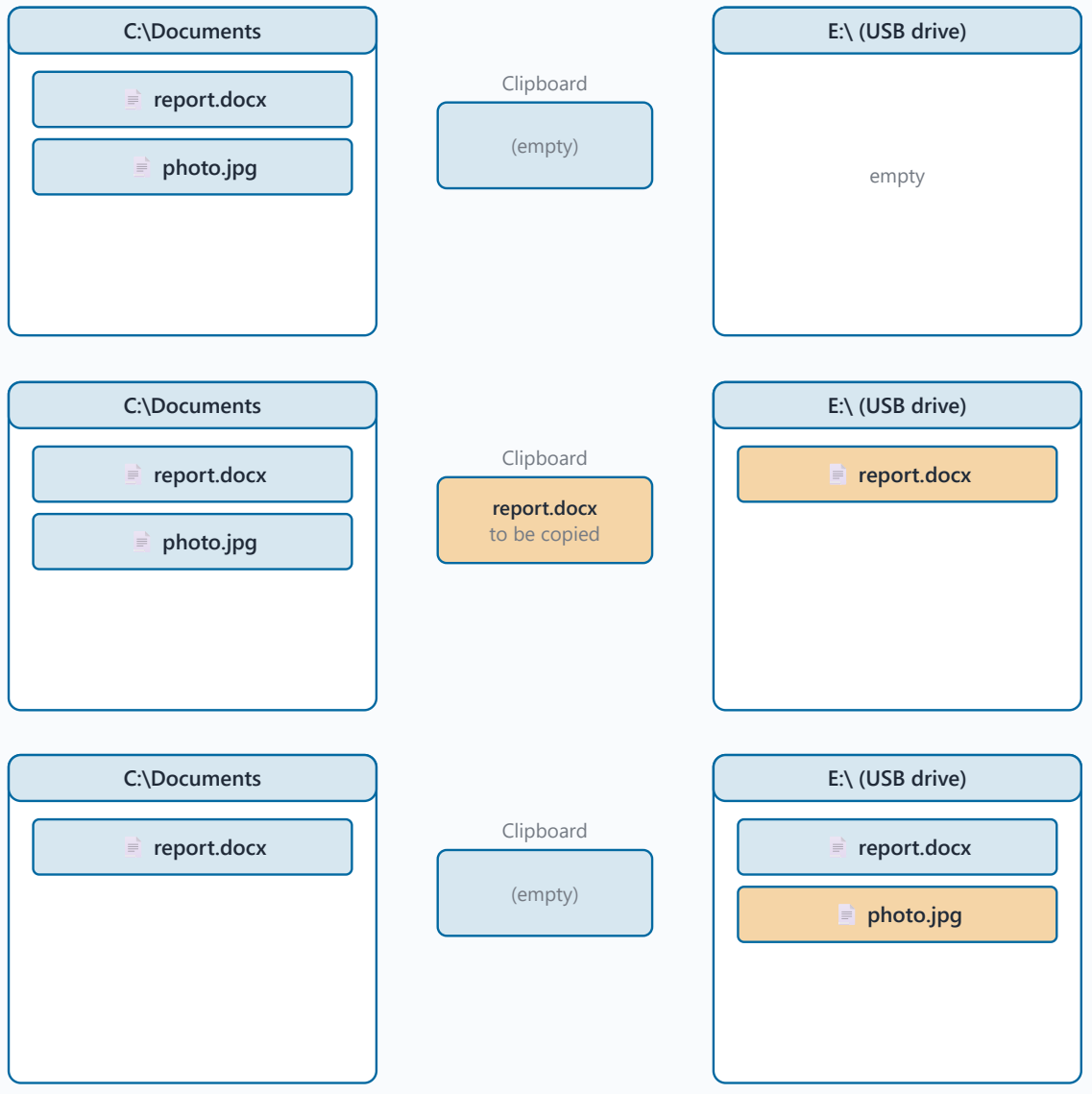


Figure 2.2.17: Three moments from copying and moving. At the top, the starting state: two files in C:\Documents and an empty USB drive. In the middle, report.docx has been copied and pasted; there is a report.docx in both locations. At the bottom, photo.jpg has been cut and pasted onto the USB drive; it has left the source and gone to the destination.

Copying has one more consequence. Because the two files are independent, a change made at the source is not reflected in the copy. In the example above, if `C:\Documents\report.docx` is opened, changed, and saved, the copy on the USB drive keeps its old content. This is useful when making a backup, but it creates confusion when you work with two versions of the same document. If it is not known which copy is current, work can be lost.

Common mistake

Moving an important file without a backup and not checking the destination. If the USB drive is pulled out during the move, or the wrong destination is chosen, the file may end up somewhere unexpected. With important files, it is safe to have a backup before moving and, after the operation, to open the destination folder and confirm that the file is there.

2.2.7.2 Drag and drop

Holding a file with the mouse and dropping it on another folder removes the need for the Copy or Cut commands. However, whether drag and drop copies or moves depends on which drives the source and destination are on.

Definition 2.2.17

Drag and drop is the operation of carrying an item to another location while the mouse button is held down and releasing it there. The default rule is this: if the source and destination are *on the same drive*, the item is moved; if they are *on different drives*, it is copied. Holding Shift while dropping forces a move, and holding Ctrl forces a copy. Holding Alt creates a shortcut to the item instead of the item itself.

There is a reason for this rule. Moving within the same drive is very fast, because the bytes of the file do not change place on the disk; only the record in the file system passes from one folder to the other. Between different drives, all the bytes must be read and written. In that case Windows takes not touching the original file, that is, copying, to be the safe option.

Example 2.2.13

At the start, `C:\Documents` holds `photo.jpg`, `report.docx`, and `note.txt`; `C:\Archive` and E:\ are empty. Four drag-and-drop operations are carried out in turn:

Step	Operation	Rule	Result
1	photo.jpg → C:\Archive	same drive	move: photo.jpg leaves Documents
2	report.docx → E:\	different drive	copy: report.docx in two places
3	note.txt → E:\, Shift held	different drive, Shift	move forced: note.txt leaves C:

Step	Operation	Rule	Result
4	E:\report.docx → C:\Archive, Ctrl held	different drive, Ctrl	copy: report.docx in three places

Table 2.2.15: Applying the drag-and-drop rules.

In the final state, `C:\Documents` holds only `report.docx`, `C:\Archive` holds `photo.jpg` and `report.docx`, and E:\ holds `report.docx` and `note.txt`. The first and third steps are moves; the second and fourth are copies. In the fourth step the result would not change even without Ctrl, because E:\ and C:\ are different drives. Ctrl only makes the intention explicit.

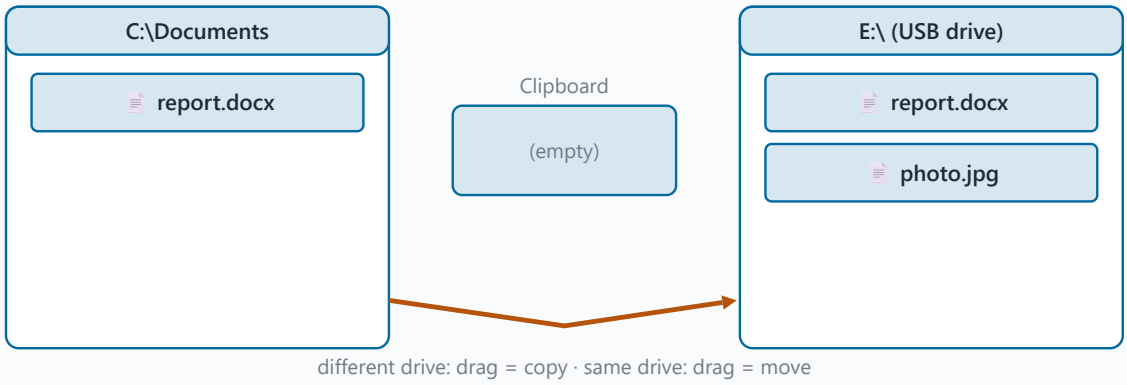


Figure 2.2.18: The drag-and-drop rule: dragging to a different drive copies, and dragging within the same drive moves.

Common mistake

Thinking that drag and drop always copies or always moves. While you drag, a small tooltip appears next to the cursor: “Move to Archive” or “Copy to E:”. Looking at this tooltip before releasing the mouse prevents an unexpected move.

2.2.7.3 Renaming and deleting

When a file or folder is selected and F2 is pressed, its name becomes editable. While extensions are visible, Windows selects only the part of the name before the extension, so the new name you type does not break the extension. Even so, take care not to change the extension (2.2.4.3). If several files are selected and F2 is pressed, they are all given the same name and numbered in order: `holiday (1).jpg`, `holiday (2).jpg`, …

Definition 2.2.18

The **Recycle Bin** is the special folder in which files deleted with Delete are held before being permanently deleted. An item in the bin returns to its original location with the *Restore* command. **Permanent deletion** deletes an item without sending it to the bin; it is done with Shift + Delete or by emptying the bin.

Figure 2.2.19 shows the two ways of deleting side by side. A file deleted with Delete goes to the Recycle Bin. Two paths lead out of it: Restore returns the file to its original folder, and Empty Recycle Bin deletes it permanently. Shift + Delete skips the bin: Windows asks for confirmation, and when confirmed, the file is permanently deleted directly.

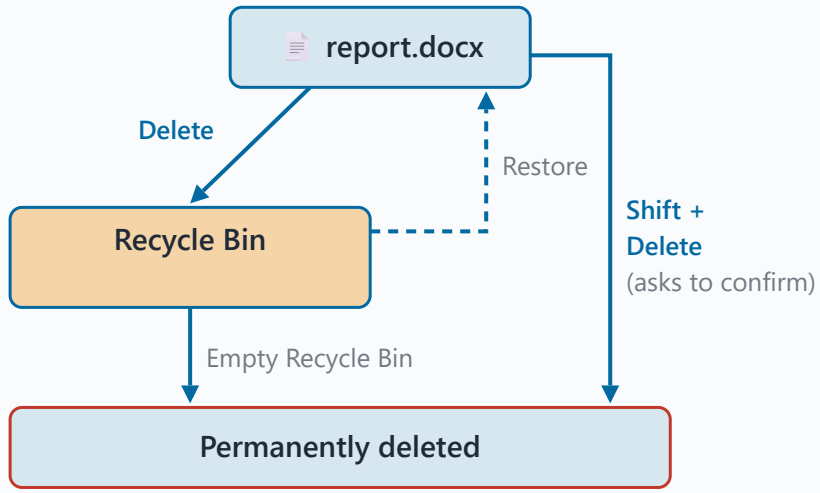


Figure 2.2.19: Delete moves the file to the Recycle Bin; from there it can be restored, or it is deleted permanently when the bin is emptied. Shift + Delete asks for confirmation and deletes the file directly and permanently.

The Recycle Bin does not protect in every case. There are three important exceptions:

- **USB drives and network locations.** Files deleted in these locations mostly do not go to the bin; they are deleted directly. In this case Windows asks “Permanently delete?”.
- **Open or locked files.** A file that an application keeps open may not be deletable. Windows gives an “the file is open in another program” warning; the application is closed first.
- **Cloud folders.** A file deleted in a synced folder such as OneDrive is also deleted from the other devices linked to the same account. The cloud’s own recycle bin offers a chance of recovery for a while.

Example 2.2.14

On the desktop (drive C:) there are `a.txt`, `b.txt`, and `c.txt`. `a.txt` is selected and Delete is pressed; `b.txt` is selected, Shift + Delete is pressed, and confirmation is given. Then `d.txt` on a USB drive is selected and Delete is pressed.

1. `a.txt` goes to the Recycle Bin. If the bin is opened and Restore is chosen, it returns to the desktop.
2. `b.txt` does not go to the bin; it is deleted permanently and cannot be brought back with Windows' own tools.
3. `c.txt` has not been touched; it stays on the desktop.
4. Because `d.txt` is on the USB drive, it does not go to the bin even with Delete. Windows asks for confirmation of permanent deletion.

In the end, only `a.txt` is in the Recycle Bin.

Common mistake

Treating the Recycle Bin as a backup. When the bin reaches a certain fullness, it automatically deletes the oldest items. Storage Sense can also clean out items older than a set time (2.2.10.1). If the disk fails, the files in the bin are lost too. The bin is only a short-term safeguard against accidental deletion.

2.2.7.4 Shortcuts and original files

Definition 2.2.19

A **shortcut** is a small file that holds the path of another file, folder, or program. Its extension is `.lnk` and its size is usually 1–2 KB. Double-clicking a shortcut opens the original item at the path it holds. There is a small arrow in the bottom-left corner of a shortcut's icon.

A shortcut makes it easier to reach a file that is opened often. The shortcut of a file sitting in a deep folder is placed on the desktop; the file itself never moves. A shortcut is created with right-click → *Create shortcut* or by dragging the file while holding the Alt key.

The most important property of a shortcut is that it contains not the file itself but only its path. This property has three consequences (Table 2.2.16).

Operation	Effect on the shortcut	Effect on the original file
the shortcut is deleted	the shortcut is gone	none; the file stays in place

Operation	Effect on the shortcut	Effect on the original file
the original file is moved to another folder	points to the old path, may fail to open	the file is at its new location
the shortcut is copied to a USB drive and opened on another computer	“target not found” error	the file stays on the first computer

Table 2.2.16: The consequences of operations on a shortcut and its original file.

Example 2.2.15

On the desktop there is a “report.pdf - Shortcut” pointing to the file `C:\Users\Sercan\Documents\report.pdf`. Three situations are examined in turn.

1. **The shortcut is deleted.** The icon on the desktop disappears. The `report.pdf` in the `Documents` folder stays where it is. Deleting a shortcut does not delete the file.
2. **A new shortcut is created, then the original file is moved to Downloads.** The shortcut still holds the path `…\Documents\report.pdf`. Because the file is no longer at that path, the shortcut may fail to open. Windows sometimes finds the file’s new location and offers to repair the shortcut, but this cannot be relied on.
3. **The shortcut is copied to a USB drive and double-clicked on another computer.** What goes to the USB drive is the 1 KB `.lnk` file, not the PDF itself. The other computer has no path `C:\Users\Sercan\Documents\report.pdf`, and a “target not found” error appears.

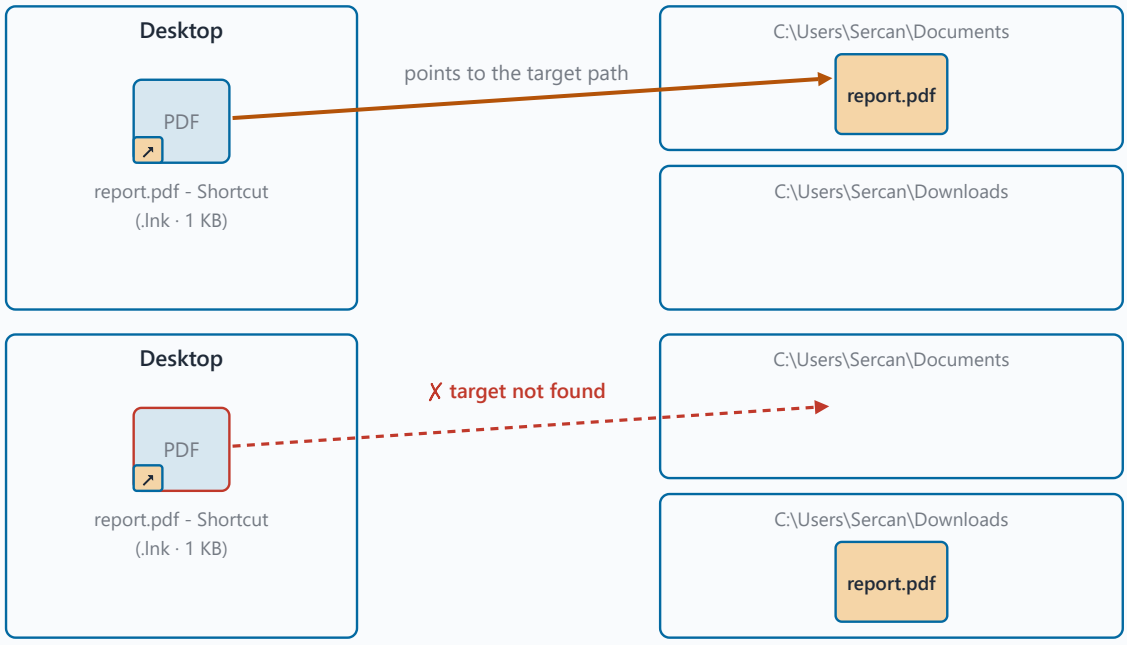


Figure 2.2.20: A shortcut and the original file. At the top, the 1 KB report.pdf - Shortcut file on the desktop points to C:\Users\Sercan\Documents\report.pdf. At the bottom, the original file has been moved to Downloads; because the shortcut points to the old path, the target cannot be found.

Common mistake

Thinking you made a backup by copying desktop shortcuts to a USB drive. If the computer fails, all that is left on the USB drive are small `.lnk` files that point nowhere. A backup copies the files themselves. Whether a file is a shortcut can be told from the arrow on the icon, from the word “Shortcut” in the Type column, or from its size of 1–2 KB.

2.2.7.5 ZIP archives

When you need to send several files of an assignment by e-mail, or upload them to a system as a single file, the files are gathered in an archive.

Definition 2.2.20

An **archive** is a file that gathers several files and folders into a single file. **Compression** is the process of reducing a file’s size by writing repetitions in the data in a shorter form. ZIP is a common format that both archives and compresses without loss; Windows creates and opens ZIP files without any extra program. Taking the files out of an archive is called **extraction**.

Lossless compression guarantees that the bytes of the extracted file are exactly the same as the original. How much compression shrinks a file depends on the content. In plain text, the same words and letter sequences repeat a lot, so text compresses well. JPEG photos and DOCX documents, however, are already compressed formats. A DOCX file is actually a ZIP archive containing XML files. Compressed data has little repetition; compressing it a second time gains almost nothing.

Example 2.2.16

Three files in `Documents\Courses` are gathered into a ZIP archive: `report.docx` (480 KB), `notes.txt` (120 KB), `photo.jpg` (2,410 KB). The steps are as follows:

1. **Selection.** While Ctrl is held down, the three files are clicked one by one. Ctrl + click adds or removes a single item from the selection; Shift + click selects all rows between two clicks. The status bar says “3 items selected.” The total size of the selection is 480 + 120 + 2,410 = 3,010 KB.
2. **Compression.** Right-click the selection and choose *Compress to → ZIP file*. In older versions of Windows 11, this command is named *Compress to ZIP file*; Microsoft’s help page gives the route “Show more options” → “Send to” → “Compressed (zipped) folder.” The archive is created in the same folder and its name comes up selected; `hw_sercan` is typed and Enter is pressed. The original files are not deleted.
3. **Result.** `hw_sercan.zip` is 2,905 KB. Table 2.2.17 shows how much each file shrank.

4. **Extraction.** Right-click the archive and choose *Extract All...* → *Extract*. A folder named `hw_sercan` , the same as the archive, is created and the three files come out into it at their original sizes.

File	Original	Compressed	Remaining ratio	Saving
notes.txt	120 KB	31 KB	25.8%	74.2%
report.docx	480 KB	468 KB	97.5%	2.5%
photo.jpg	2,410 KB	2,405 KB	99.8%	0.2%
Total	3,010 KB	2,904 KB (+ archive info)	ZIP 2,905 KB; saving ≈ 3.5%	

Table 2.2.17: Compression results of the files.

The remaining ratio is the compressed size divided by the original size: for the notes, 31 ÷ 120 ≈ 0.258, that is, 25.8%. The text file has shrunk to a quarter. The DOCX and JPEG have hardly shrunk at all. The compressed sizes add up to 31 + 468 + 2,405 = 2,904 KB. The ZIP file is slightly larger than this because it also holds the names and properties of the files inside. The saving for the whole archive is (3,010 − 2,905) ÷ 3,010 ≈ 3.5%. In this example, the archive's real benefit is not shrinking but gathering three files in one package.

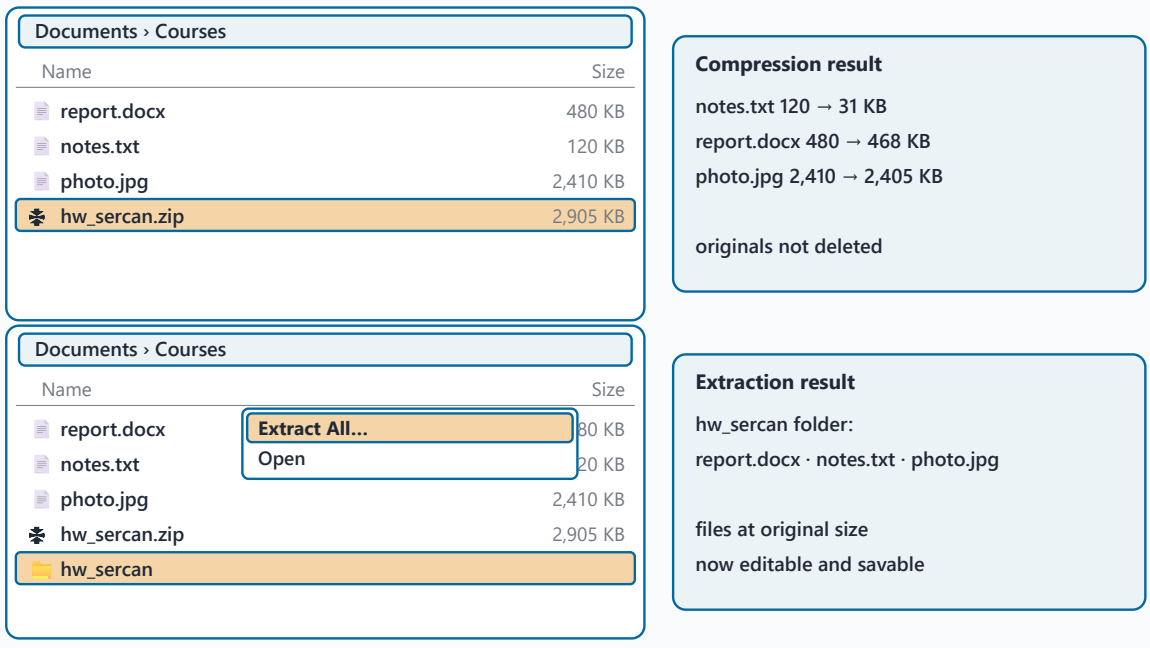


Figure 2.2.21: A ZIP archive. At the top, three files are gathered in the archive hw_sercan.zip (2,905 KB); notes.txt has shrunk from 120 KB to 31 KB, and report.docx and photo.jpg have hardly shrunk. At the bottom, Extract All has created the hw_sercan folder, and the files have been extracted at their original sizes.

Common mistake

Double-clicking a document inside a ZIP and editing it. Double-clicking a ZIP shows the inside of the archive like a folder, but a document opened from there is a copy in a temporary folder. Even if the change is saved in Word, it is written to the temporary copy; the file in the ZIP does not change, and the effort is easily lost. Editing is always done first on an extracted copy.

2.2.7.6 File and folder properties

When you right-click a file and choose *Properties* (or press Alt + Enter with the file selected), a window opens that shows the file's details (Figure 2.2.22).

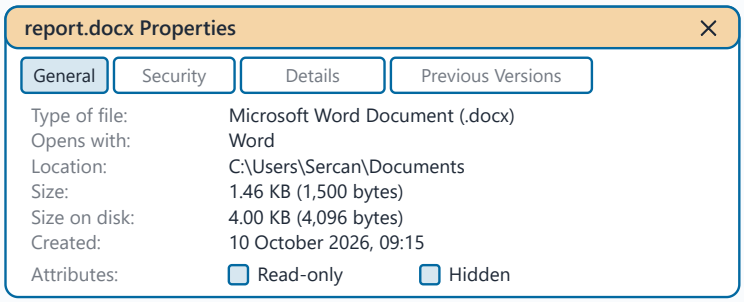


Figure 2.2.22: The General tab of the Properties window of report.docx: type, opens with, location, size, size on disk, created date, and the Read-only / Hidden attributes. At the top are the Security, Details, and Previous Versions tabs.

The General tab contains the following information:

- **Type and app**: the file's format (Microsoft Word Document, .docx) and the application that opens when it is double-clicked. The *Change* button next to it changes this application.
- **Location**: the full path of the folder the file is in.
- **Size** and **size on disk**: the size of the content and the space reserved on the disk. Why these two numbers differ is explained in the example below.
- **Dates**: the created, modified, and accessed dates.
- **Attributes**: *Read-only* prevents the file from being changed; *Hidden* hides the file from the list (2.2.6.2).

The Security tab shows the file's owner and which users can read or change the file. The Previous Versions tab lists old copies of the file if File History or a restore point is turned on (2.2.10.6).

Definition 2.2.21

A file system divides the disk into fixed-size blocks. In NTFS the size of these blocks is usually 4 KB (4,096 bytes). Every file larger than a few hundred bytes takes at least one block and uses a whole number of blocks. (Very small files can be kept in NTFS's own record table; their size on disk appears as 0.) The **size on disk** is therefore calculated as follows:

> size on disk = ⌈size ÷ 4,096⌉ × 4,096 bytes

The ⌈ ⌉ sign here means rounding up.

Example 2.2.17

The file in Figure 2.2.22 is 1,500 bytes. Why does Windows show its size as "1.46 KB" and its size on disk as "4.00 KB"?

1. Windows calculates KB as 1,024 bytes: 1,500 ÷ 1,024 ≈ 1.46. The size appears as 1.46 KB.
2. How many blocks does the file take? 1,500 ÷ 4,096 ≈ 0.37; rounded up, 1 block.
3. The size on disk is 1 × 4,096 = 4,096 bytes = 4.00 KB. The remaining 2,596 bytes of the block are empty, but no other file can use them.

The same rule holds for larger files. A 10,000-byte file takes 10,000 ÷ 4,096 ≈ 2.44 → 3 blocks; its size on disk is 3 × 4,096 = 12,288 bytes. With many small files the difference grows: the total size of 100 files of 1,500 bytes each is 150,000 bytes, but on disk they take 100 × 4,096 = 409,600 bytes (400 KB).

Common mistake

Thinking that the Read-only attribute also protects a file from deletion. Read-only prevents the file's content from being changed and saved under the same name. The file can still be copied, saved under a different name, and deleted. Attributes are a layer of reminders, not a security measure.

2.2.8 Settings

Most of the options that determine how Windows behaves are gathered in a single application: Settings. The screen resolution, the Wi-Fi network you connect to, installed applications, user accounts, the keyboard layout, and updates are all managed from here.

2.2.8.1 The Settings app

Settings opens with Win + I or with the gear icon in the Start menu. The left column holds the categories, and the right shows the pages of the selected category. Category names may differ a little between Windows versions, but the main areas stay the same. Table 2.2.18 lists the categories most used in these notes.

Category	Settings included
System	display, sound, notifications, power, storage, recovery, about
Bluetooth & devices	mouse, keyboard, printers, and connected devices
Network & internet	Wi-Fi, Ethernet, VPN, and data usage
Personalization	background, themes, Start, and taskbar
Apps	installed apps, default apps, startup
Accounts	sign-in options, Windows backup, other users
Time & language	date and time, language and region, keyboard
Accessibility	text size, magnifier, color filters, screen reader, captions
Privacy & security	app permissions and Windows Security
Windows Update	updates, restarts, and update history

Table 2.2.18: The main categories of the Settings app.

If you do not know in which category a setting is, use the search box at the top left of the Settings window. This box works in the same way as the search in the Start menu, but it searches only among settings. In these notes the location of a setting is written as “Settings → System → Display.” This notation means opening the Settings app, then the System category, then the Display page, in that order.

2.2.8.2 Wi-Fi, printers, and display

The first three settings made on a new computer or in a new place are usually connecting to a network, adding a printer, and adjusting the display.

Connecting to Wi-Fi. Click the network icon at the right of the taskbar or press Win + A; the Quick settings panel opens. The arrow next to the Wi-Fi button lists nearby networks. Select a network, press *Connect*, and enter the network’s password. If “Connect automatically” is checked, the computer recognizes this network by itself next time.

Adding a printer. Follow Settings → Bluetooth & devices → Printers & scanners → *Add device*. A printer on the same network or connected by USB is usually found automatically, and its driver is installed by itself. If the printer’s driver cannot be found, it is downloaded from the manufacturer’s website.

Display resolution and scale. The Settings → System → Display page has two separate settings. *Display resolution* determines how many pixels the image is made of. This value should stay at the screen's native resolution, the value Windows marks "(Recommended)." If a lower resolution is chosen, the image becomes blurry. *Scale* adjusts the size of text and icons. If the text seems small, you do not lower the resolution; you raise the scale.

Definition 2.2.22

Scale is the percentage that determines how many times larger Windows draws text, icons, and windows. At 100% scale an interface element is drawn 1 pixel to 1 pixel; at 150% the same element is drawn 1.5 times larger. The *logical* area available to applications is the physical resolution divided by the scale.

Example 2.2.18

The screen of a 15.6-inch laptop is 1920 × 1080 pixels. What logical area is left for applications when the scale is 100%, 125%, and 150%?

1. 100%: 1920 ÷ 1 = 1920, 1080 ÷ 1 = 1080. The logical area is 1920 × 1080; text is at its smallest.
2. 125%: 1920 ÷ 1.25 = 1536, 1080 ÷ 1.25 = 864. The logical area is 1536 × 864.
3. 150%: 1920 ÷ 1.5 = 1280, 1080 ÷ 1.5 = 720. The logical area is 1280 × 720; text is large and little content fits on the screen.

In all three cases all 1920 × 1080 physical pixels of the screen are used, and the image stays sharp. Lowering the resolution to 1280 × 720 enlarges the text by the same ratio, but because each logical pixel is spread over 1.5 physical pixels, the image becomes blurry. The right way to enlarge text is scale.

Common mistake

Signing in to important accounts such as banking or school accounts on a public Wi-Fi network (a café, an airport). Others can watch the traffic on these networks, or a fake access point can imitate the name of the real network. Important operations are done on a trusted network or over mobile data.

2.2.8.3 User accounts

A computer can be used by several people. Windows opens a separate **user account** for each person. Each account has its own desktop, Documents folder, settings, and password. Separate accounts keep personal files and settings apart between users. Other standard users cannot access a user's

`C:\Users\Ayse` folder.

Accounts are classified by two separate questions. The first question is where the credentials are managed:

- **Local account**: the user name and password are kept only on that computer. It is not tied to another device.
- **Microsoft account**: an online account opened with an e-mail address. It can link supported settings, OneDrive files, and Microsoft Store apps across devices; if the password is forgotten, it can be recovered online.

The second question is the account's authority:

- **Standard user**: does everyday work and changes their own files and settings. Administrator permission is needed for tasks such as installing programs and changing system-wide settings.
- **Administrator**: installs and removes apps, manages the other accounts, and can change all system settings.

When a task that needs administrator permission is started, Windows dims the screen and shows a confirmation window. This feature is called **User Account Control** (UAC). Figure 2.2.23 compares what UAC asks for in the two account types. If a standard user wants to install an app, an administrator's name and password are requested. If an administrator does the same task, only a Yes / No confirmation is asked.

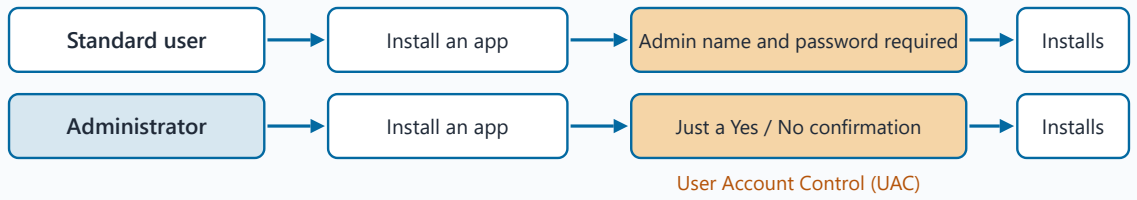


Figure 2.2.23: User Account Control. When a standard user wants to install an app, an administrator name and password are requested; for the same task an administrator gives only a Yes / No confirmation. In both cases the app is installed after confirmation.

There is a reason confirmation is requested even on an administrator account. If a malicious program tries to install itself in the background, the UAC window appears. If the user is not installing anything at that moment, this window is a warning sign, and No is chosen. So it is a safe principle to use a standard account for everyday work and to turn on administrator authority only when needed.

Signing out closes the open apps and ends the user's session. Another user, in turn, signs in on the same computer with their own account. A new account is added on the Settings → Accounts → Other users page; the account type is chosen on the same page with *Change account type*.

2.2.8.4 App management and permissions

From an app's arrival on the computer to its departure, several decisions are made: where it is installed from, which permissions it is given, whether it runs when you sign in, what to do when it causes trouble, and finally how it is removed.

- **Install source.** Apps are obtained from the Microsoft Store or from the vendor's official site. "Free" versions on unfamiliar sites often carry malware.
- **Uninstalling.** On the Settings → Apps → Installed apps page, click the three dots next to the app and choose *Uninstall*. This is the right way; deleting an app's shortcut or folder does not uninstall it.
- **Startup apps.** Some apps run by themselves when you sign in. This list is limited on the Settings → Apps → Startup page or in Task Manager (2.2.10.4).
- **Permissions.** Camera, microphone, location, and file access are controlled per app.
- **Repair and reset.** For supported apps, the *Repair* and *Reset* buttons are on the Installed apps → Advanced options page. Repair tries to fix the app without touching its data; Reset deletes the app's data and returns it to the state it was in when first installed.
- **Updates.** Apps are kept up to date for security and compatibility. Store apps update automatically; other apps usually show their own update notifications.

Permissions determine how much an app can access personal data and devices. Figure 2.2.24 shows the camera permission page. The Camera app and Teams are allowed to use the camera, and a game is turned off. A game does not need the camera. Giving an app only the permissions it needs to do its job prevents data from being collected needlessly.

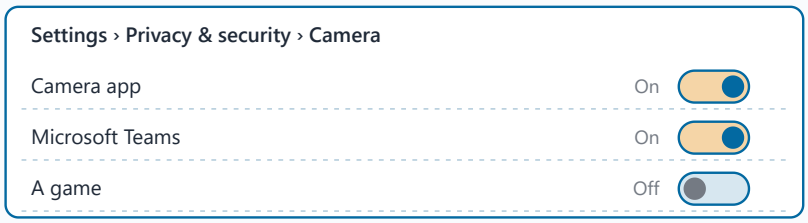


Figure 2.2.24: The Settings → Privacy & security → Camera page: permission is on for the Camera app and Microsoft Teams, and off for a game.

Common mistake

Deleting an app's folder in Program Files to uninstall it. Even if the folder is deleted, the app's registry entries, shortcuts, and background services remain on the system, and Windows still thinks the app is installed. Apps are uninstalled from the Installed apps page in Settings.

2.2.8.5 Default apps

Definition 2.2.23

A **default app** is the program Windows chooses automatically when a particular file type or link type is opened. For example, the default app for `.pdf` files may be Microsoft Edge.

A default app is the user-visible face of the association described in 2.2.4.3. Figure 2.2.25 shows this mapping for three files. When a file is double-clicked, Windows first looks at the extension, then launches the app associated with that extension.

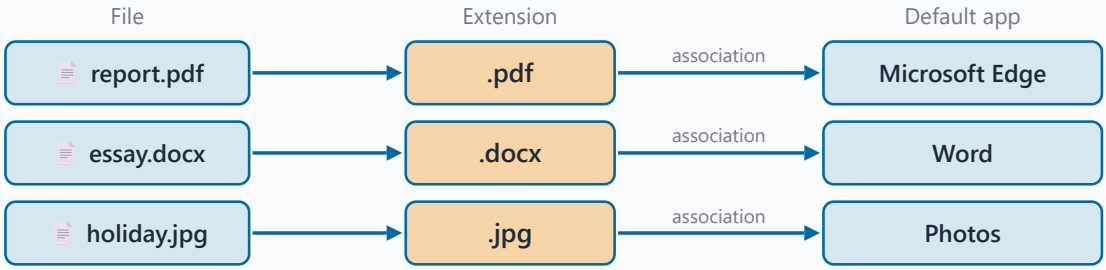


Figure 2.2.25: The default app mapping by extension: report.pdf (.pdf) → Microsoft Edge, homework.docx (.docx) → Word, holiday.jpg (.jpg) → Photos.

There are two ways to change a default. For a permanent change, search for the file type (for example `.pdf`) on the Settings → Apps → Default apps page and choose the new app. For a one-time change, right-click the file, choose *Open with*, and pick another app from the list. If "Always" is chosen in the Open with window, that choice becomes permanent as well.

Example 2.2.19

The default app for `.pdf` files is Edge. In which app does `homework.pdf` open in the following operations?

Step	Operation	App that opens	Default
1	`homework.pdf` is double-clicked	Edge	Edge
2	right-click → Open with → Acrobat Reader ("Always" is not chosen)	Acrobat Reader	Edge
3	double-clicked again	Edge	Edge
4	In Settings → Apps → Default apps, Acrobat is chosen for .pdf, then the file is double-clicked	Acrobat Reader	Acrobat Reader

The choice in the second step is one-time and does not change the default. That is why the file opens in Edge again in the third step. In the fourth step the default is changed by file type, so all double-clicks from then on open Acrobat Reader.

Common mistake

Trying to run an unrecognized file type with the thought "maybe some program will open it." For a file with an unknown extension, especially if it came as an e-mail attachment, the source is verified before it is opened. When Windows asks "How do you want to open this file?", find out what the file is instead of choosing an app at random.

2.2.8.6 Accessibility

Not everyone who uses a computer sees, hears, or uses a mouse in the same way. Accessibility settings adapt Windows to users with different needs. These settings are not only for users with disabilities. Anyone who works at a screen for long periods, watches a video in a noisy place, or cannot use one hand can benefit. The settings are gathered in the Settings → Accessibility category (Table 2.2.19).

Feature	What does it do?	Who benefits?
Screen reader (Narrator)	reads the text and controls on the screen aloud; turned on and off with Win + Ctrl + Enter	blind users
Magnifier	enlarges part or all of the screen; opened with Win + plus (+)	low-vision users, instructors who show small details
Contrast and color	contrast themes make text stand out; color filters improve color distinction	low-vision and color-blind users
Live captions	shows speech playing on the computer or coming in through the microphone as text on the screen	deaf and hard-of-hearing users, people watching video in a quiet place
Voice access	controls supported operations with voice commands	users who cannot use a mouse and keyboard
Keyboard and mouse	Sticky Keys, Filter Keys, a large and colored mouse pointer	users with limited movement

Table 2.2.19: The main accessibility features.

Sticky Keys is a good example. A three-key shortcut such as Ctrl + Shift + Esc cannot be pressed with one hand or one finger. With Sticky Keys on, modifier keys such as Ctrl, Shift, and Alt are treated as held down. The user presses the keys one at a time, in order, and Windows interprets them as if pressed together. Filter Keys, on the other hand, ignores brief or repeated unintended key presses. It prevents a trembling hand from typing the same letter three times.

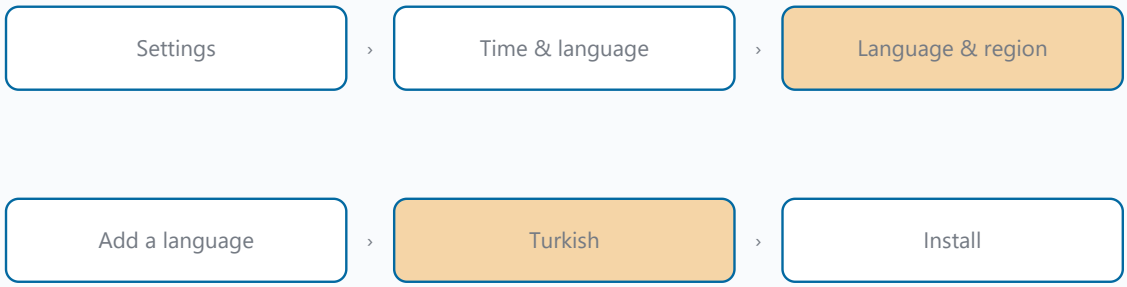
2.2.8.7 Turkish keyboard and language

The letters printed on the keyboard and the meaning the computer gives to the keys are separate things. Which key produces which letter is determined by the **keyboard layout** setting. Two layouts are used in Turkey. **Turkish Q** is the most common layout; it adds the Turkish letters to the letter arrangement of the English keyboard. **Turkish F** is an alternative layout designed around the letter frequencies of Turkish and developed for fast typing. On the F keyboard the most frequently used letters are placed in the middle row where the fingers rest.

Example 2.2.20

A computer has only the Turkish Q keyboard. You want to add and use Turkish F. The steps are as follows:

1. Press Win + I to open Settings. The language and keyboard settings are in the *Time & language* category; from there, go to the *Language & region* page.
2. If Turkish is not in the list, choose it with *Add a language* and install it. Turkish can also be used as the display language.
3. Open the three dots next to Turkish → *Language options*. In the Keyboards section, choose Turkish F with *Add a keyboard*. The active keyboard does not change at this point; Turkish F has only been added to the list.
4. Win + Space switches between the installed keyboard layouts. The active layout appears at the right of the taskbar (for example TR F).
5. Pressing Win + Space once more goes back to Turkish Q. Alt + Shift is also a classic shortcut for switching language or layout, and it can be changed in settings.



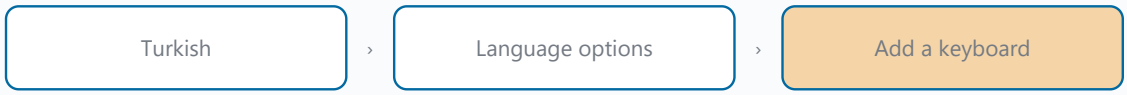


Figure 2.2.26: The route for adding a Turkish keyboard and language: Settings › Time & language › Language & region; Add a language › Turkish › Install; Turkish › Language options › Add a keyboard.

Common mistake

Pressing Win + Space or Alt + Shift by accident, changing the layout, and thinking the keyboard is "broken." The symptom is that keys type characters different from the letters printed on them, such as a semicolon or a different letter appearing in place of the one you pressed. Looking at the language indicator on the taskbar and, if necessary, returning to the right layout with Win + Space solves the problem. In a password box this is especially annoying, because the characters typed are not visible.

2.2.9 Security

The security of a computer has several layers. Signing in controls who can use the computer. Antivirus and the firewall stop malware and attacks from the network. The user's habits also determine whether all these measures will work.

2.2.9.1 Windows Hello and sign-in security

Definition 2.2.24

Windows Hello is a way of signing in to Windows with a PIN, face recognition, or a fingerprint instead of a password. In this method the PIN or **biometric** data (face, fingerprint) never leaves the device. It unlocks a private key stored in a security chip inside the device (the TPM); authentication is done with this key.

Figure 2.2.27 shows the steps of the verification. The user enters the PIN or shows their face to the camera. This gesture unlocks the private key inside the TPM. The account server sends a challenge to the device, and the device signs this challenge with the private key. The server verifies the signature with the public key previously registered with it. Not the PIN but only a digital signature travels over the network.

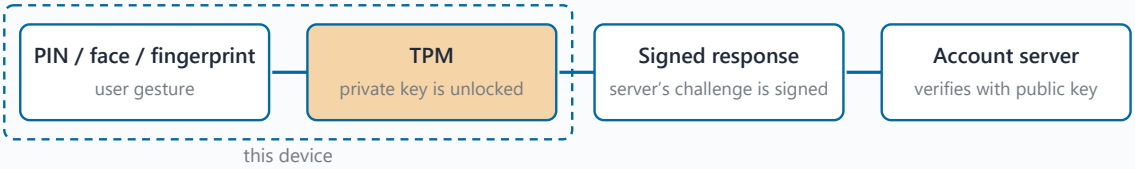


Figure 2.2.27: Windows Hello. A PIN or biometric gesture unlocks the private key in the TPM on this device; the key signs the server's challenge, and the server verifies the signature with the public key. The PIN and biometric data never leave the device.

This design explains why a short PIN can be safer than a long password. A password belongs to the account itself; if it is stolen, the account can be entered from anywhere in the world. A PIN is tied to a single device. A stolen PIN is of no use to an attacker who does not have that device. Moreover, the TPM notices repeated wrong PIN attempts and locks the device for a while.

Example 2.2.21

How many possible values does a 4-digit PIN have? A 6-digit PIN? How do these compare with an 8-character password that may contain uppercase letters, lowercase letters, and digits?

1. Each digit is one of 10 digits. For 4 digits there are $10 \times 10 \times 10 \times 10 = 10^4 = 10{,}000$ possibilities.
2. For 6 digits there are $10^6 = 1{,}000{,}000$ possibilities.
3. In a password, each character is one of 26 lowercase letters, 26 uppercase letters, and 10 digits, that is, 62 characters. For 8 characters there are $62^8 \approx 2.18 \times 10^{14}$ possibilities.

By the number of possibilities, the password looks far stronger. But an attacker can try a PIN only at that device and only a limited number of times; the TPM imposes a waiting period after a few wrong attempts. A password, on the other hand, can be tried online from a leaked list. The strength of a PIN comes not from the number of possibilities but from its being tied to the device.

The other parts of sign-in security are these:

- **Screen lock.** When leaving the computer, lock the session with Win + L.
- **Multi-factor authentication.** When signing in to an account, a second proof is requested in addition to the password: a code sent to the phone or a confirmation in an authenticator app. Even if the password is stolen, the account cannot be entered without the second proof.
- **Recovery.** How to get back into the account if the PIN is forgotten or the device is changed should be arranged in advance: a current phone number, a backup e-mail address, or recovery codes.

These options are on the Settings → Accounts → Sign-in options page.

2.2.9.2 Windows Security

Windows has protection against malware built in. The tools of this protection are gathered in the **Windows Security** app. The app is opened through Settings → Privacy & security → Windows Security or by searching for "Windows Security" in the Start menu. Table 2.2.20 summarizes the main sections.

Section	Job
Virus & threat protection	scans for and blocks malware; quick scan and real-time protection are here
Firewall & network protection	controls connections coming in and going out over the network; blocks unauthorized access
App & browser control	warns about dangerous downloads and malicious sites with SmartScreen
Device security	shows the status of hardware-based protections such as Secure Boot and the TPM
Account protection	shows the Microsoft account and Windows Hello sign-in protections

Table 2.2.20: The sections of Windows Security.

The software that provides virus and threat protection is called **Microsoft Defender**. Defender comes with Windows and stays active unless another antivirus is installed. Its most important feature is **real-time protection**: when a file is downloaded, opened, or run, Defender scans it at that moment. This protection should be kept on, and Defender's definition updates should not be interrupted. Newly appearing malware can be recognized only with up-to-date definitions.

A **firewall** is a gatekeeper that filters the traffic between the computer and the network according to rules. For example, it blocks a connection request that comes from outside and that no application is expecting. **SmartScreen** checks whether downloaded files and visited sites are on known dangerous lists. It is what shows the "Windows protected your PC" warning when an unrecognized program is about to be run.

Two concepts named in the **Device security** section form the hardware layer of protection. **Secure Boot** allows only software with a verified signature to load while the computer starts; it stops malware that tries to run before the operating system. The **TPM** (Trusted Platform Module) is a security chip that stores keys and passwords securely. Windows Hello and BitLocker use this chip.

Common mistake

Turning off real-time protection in order to install a program. If an installer says "turn off your antivirus," it is most likely malicious. Software from a trusted source can be installed with Defender on. Once the protection is turned off, Windows turns it back on by itself after a while; every download in between goes unchecked.

2.2.9.3 Safe everyday use

However good the tools are, most of security depends on the user's habits. Six habits defeat most ordinary attacks:

- **Lock.** When leaving the computer, lock the screen with Win + L.
- **Updates.** Keep Windows and apps up to date. Most updates close known security holes.
- **Standard account.** Do everyday work with a standard account; use administrator authority only when needed.
- **Downloads.** Get software from the Microsoft Store or from the vendor's official site.
- **Backups.** Back up important files regularly and to a separate location (2.2.10.5).
- **Suspicious files.** Do not open attachments and executable files from unknown sources.

These habits complement one another. For example, even if a malicious attachment is opened, if the user is on a standard account, UAC asks for an administrator password for the system change, and the attack may stop there. Even if an attack succeeds and encrypts the files, an up-to-date backup in a separate location keeps the loss to a minimum.

2.2.10 Maintenance

A computer fills up, slows down, and falls out of date over time. The disk swells with temporary files, the number of programs that run at startup grows, an app freezes, an update causes trouble. Regular maintenance prevents most of these problems in the first place. For those that cannot be prevented, Windows has recovery tools.

2.2.10.1 Storage management

When the disk is full, Windows cannot install updates, apps cannot write temporary files, and the computer slows down. What the space on the disk is spent on can be seen on the Settings → System → Storage page. This page divides the space into categories such as apps, system files, documents, and temporary files (Figure 2.2.28).

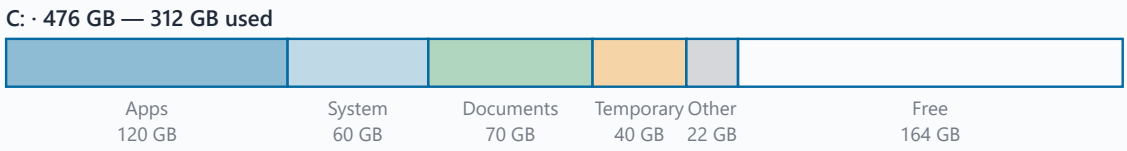


Figure 2.2.28: The distribution of space on drive C: (476 GB, 312 GB used) by category: Apps 120 GB, System 60 GB, Documents 70 GB, Temporary 40 GB, Other 22 GB, Free 164 GB.

Example 2.2.22

For the drive in Figure 2.2.28, let us calculate the usage ratio, the share of each category, and what the free space becomes once the temporary files are cleaned.

1. The sum of the categories: 120 + 60 + 70 + 40 + 22 = 312 GB. Free space: 476 − 312 = 164 GB. The numbers are consistent with the figure.
2. Usage ratio: 312 ÷ 476 ≈ 0.655, that is, 65.5%. Free space is 34.5%.
3. The largest part of the used space is apps: 120 ÷ 312 ≈ 38.5%. Temporary files take 40 ÷ 312 ≈ 12.8%.
4. If the temporary files are deleted, used space falls to 312 − 40 = 272 GB and free space rises to 164 + 40 = 204 GB. The share of free space becomes 204 ÷ 476 ≈ 42.9%.

On this drive, the first category to clean is temporary files, because deleting them loses no data. If more space is needed, the next step is to uninstall unused apps.

The Storage page offers two helper tools:

- **Storage Sense**: when turned on, it automatically cleans temporary files and, according to the rules you set, old items in the Recycle Bin. For example, it can be set to delete files that have stayed in the bin for more than 30 days. With this rule, a file deleted on 1 October 2026 is also removed from the bin after 31 October 2026.
- **Cleanup recommendations**: lists large or long-unopened files, unused apps, and temporary files. The user decides for themselves what is deleted.

Common mistake

Deleting every large file in the list to free up space. A file in the cleanup recommendations may have been suggested because it has not been opened for a long time; that does not mean it is unnecessary. An old project or the photos of a past term may show up in this list. Before deleting, make sure the file is not needed; if you cannot be sure, move the file to an external drive.

2.2.10.2 Optimizing drives

An HDD and an SSD read data in very different ways, so the maintenance applied to each is also different. The Windows **Defragment and Optimize Drives** tool is opened by searching for “defrag” in the Start menu. The tool recognizes the type of each drive and applies the suitable operation (Table 2.2.21).

Drive	Maintenance Windows applies	Why?
HDD	defragmentation	placing the pieces of fragmented files side by side

Drive	Maintenance Windows applies	Why?
SSD	TRIM / optimization	telling the SSD controller which blocks are unused

Table 2.2.21: Maintenance by drive type.

Definition 2.2.25

If the blocks of a file sit in scattered places on the disk rather than one after another, the file is **fragmented**. **Defragmentation** is the process of rearranging the blocks of each file so that they sit side by side on the disk. **TRIM** is the operating system telling the SSD which blocks of deleted files are no longer in use.

Fragmentation arises by itself as files are deleted and new files are written. If the gap a deleted file leaves is small, part of a new file is written there and the rest in another gap. On an HDD this is a problem, because the read head has to move mechanically to go from one piece to the next. Each jump takes a few milliseconds.

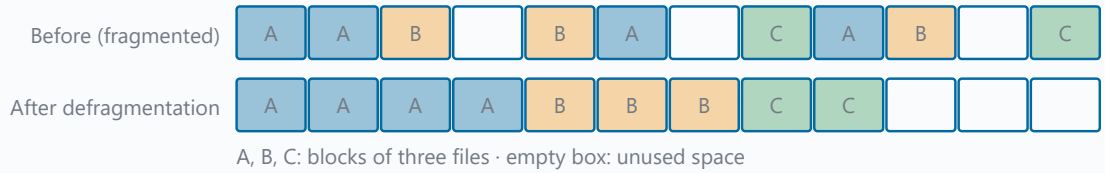


Figure 2.2.29: An HDD before and after defragmentation. First, the blocks of files A, B, and C are scattered with empty blocks between them. After defragmentation, the blocks of each file are side by side (AAAA BBB CC) and the free space is gathered at the end.

Example 2.2.23

The disk in Figure 2.2.29 has 12 blocks. Before defragmentation the sequence is A A B _ B A _ C A B _ C (_ is an empty block). How many pieces is each file in? If a jump to each piece takes an average of 9.2 ms, how much time is spent on positioning to read the three files?

1. File A is in blocks 1, 2, 6, and 9. Blocks 1 and 2 are side by side and the others are separate: 3 pieces.
2. File B is in blocks 3, 5, and 10: 3 pieces. File C is in blocks 8 and 12: 2 pieces.
3. There are 3 + 3 + 2 = 8 pieces in total. For each piece the head repositions: 8 × 9.2 = 73.6 ms.
4. After defragmentation each file is a single piece: 3 positionings, 3 × 9.2 = 27.6 ms.

Even in this small example the positioning time falls to a little over a third. On an HDD with thousands of fragmented files, the difference reaches seconds.

An SSD, on the other hand, has no moving head. Every block is read at the same speed wherever it is. So fragmentation does not slow an SSD down, and defragmentation is unnecessary. Moreover, the many writes made during defragmentation needlessly wear an SSD, which has a limited write lifetime. What an SSD really needs is TRIM. When a file is deleted, the SSD does not know it by itself. If no TRIM notification arrives, the SSD still thinks the blocks holding deleted data are full, and on new writes it must first clear those blocks; writing slows down.

Common mistake

Frequently defragmenting an SSD by hand with classic HDD logic. Windows recognizes the drive type and chooses the right operation by itself: it defragments the HDD and applies TRIM to the SSD. The tool is scheduled by default to run weekly. It is enough not to turn this setting off.

2.2.10.3 System Information and Task Manager

To solve a problem, you first need to know what is on the computer and what is running at that moment. Windows offers two tools for this.

System Information shows the details of the hardware, the operating system, the components, and the resources: the processor model, the amount of RAM, the Windows version, the BIOS/UEFI version, device drivers. It is opened by searching for “System Information” in the Start menu. A shorter summary is on the Settings → System → About page. You can tell here whether the computer meets a piece of software’s system requirements, and when someone asks for support, the computer’s specifications are read from here.

Definition 2.2.26

Task Manager is the Windows tool that shows running applications and processes and their processor (CPU), memory, disk, and network use, and ends a process when necessary. It opens directly with Ctrl + Shift + Esc, from the Win + X menu, or by right-clicking the taskbar.

A **process**, mentioned here, is a program that is running. An application can consist of several processes; browsers, for example, open a separate process for each tab. Task Manager shows in columns how many resources each process uses (Figure 2.2.30).

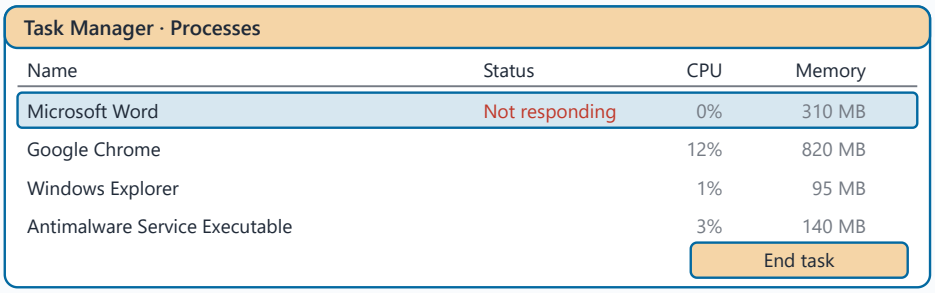


Figure 2.2.30: The Processes page of Task Manager. Microsoft Word, whose status is "Not responding," is selected; the CPU and Memory columns show each process's resource use. At the top right is the End task button.

Task Manager consists of several pages. Each page answers a different question (Table 2.2.22).

Page	What it shows	When to look
Processes	apps and background processes; CPU, memory, disk, and network percentages	when the computer slows down or an app freezes
Performance	live graphs of CPU, memory, disk, network, and GPU	to see which hardware is the bottleneck
App history	resource use of apps over time	to find which app uses a lot of network or CPU
Startup apps	programs that run when you sign in, and their impact	when startup gets slow
Users	signed-in users and their resource use	on a shared computer
Details / Services	process IDs (PID) and Windows services	for advanced troubleshooting

Table 2.2.22: The Task Manager pages.

High resource use is not always a problem. Windows Update, a virus scan, or the building of the search index can use the CPU and disk heavily for a short time. Such loads pass in a few minutes. Constant 100% disk or memory use, however, is a typical cause of slowness. When memory fills up, Windows writes some of the data in RAM to the disk and reads it back when needed; because the disk is thousands of times slower than RAM, the computer slows down noticeably.

Example 2.2.24

On a computer with 8 logical processors, a program has entered an infinite loop and is keeping a single core completely busy. What does Task Manager show in the CPU column for this program? What if the same happened on a computer with 4 logical processors?

1. The CPU percentage is computed against the whole processor capacity. Each of the 8 logical processors is 100 ÷ 8 = 12.5% of the capacity.
2. So the program that fully uses one processor appears at about 12.5%.
3. On 4 logical processors the same program appears at 100 ÷ 4 = 25%.

The remark “the program uses only 12.5%, so it is no problem” can therefore be misleading. The program consumes its own core completely. In the per-core graphs on the Performance page, you can see one core sitting at the top the whole time.

Common mistake

Assuming that every process with an unrecognized name is malicious and ending it or deleting its file. Names such as “Antimalware Service Executable” look suspicious at first glance, but this process is Microsoft Defender itself. An unknown process is not ended or deleted before its name has been looked up in trusted sources. High resource use alone is not evidence of malware either.

2.2.10.4 Frozen apps and startup apps

The two situations in which Task Manager is used most often are closing a frozen app and disabling programs that slow down startup. Let us follow both step by step on an example.

Example 2.2.25

Word does not respond to clicks. The computer also takes a long time to start. In Task Manager, opened with Ctrl + Shift + Esc, the steps are as follows:

Step	Page	Operation	Result
1	Processes	Task Manager opens	Word’s status is “Not responding”; CPU 0%, memory 312.4 MB
2	Processes	The Word row is selected and you wait a few seconds	if Word does not recover, go to the next step
3	Processes	*End task*	the Word process closes and returns its memory; Edge, Spotify, and Windows Explorer remain in the list
4	Startup apps	page opens	OneDrive, Spotify, Teams… are listed; Spotify’s startup impact is “High”
5	Startup apps	Spotify is selected, *Disable*	Spotify’s status is “Disabled”
6	Processes	the computer is restarted	Spotify does not open by itself and is not in the list

Table 2.2.23: A frozen app and a startup app in Task Manager.

The wait in the second step is important. “Not responding” does not always mean frozen; the app may be trying to finish a long operation, such as saving a large document. In the third step the Word process is closed by force. Changes made after the last save may be lost; when Word is reopened, it may offer its own automatic recovery copy.

When the task is ended, the memory use in the list changes as well. The total memory of the four processes is 312.4 + 845.2 + 198.7 + 96.3 = 1,452.6 MB. When Word's 312.4 MB is given back, the remaining three processes use 1,140.2 MB.

In the fifth step Spotify is not uninstalled; it simply does not start at sign-in. When needed, it is opened by hand from the Start menu. The same list is also managed on the Settings → Apps → Startup page.

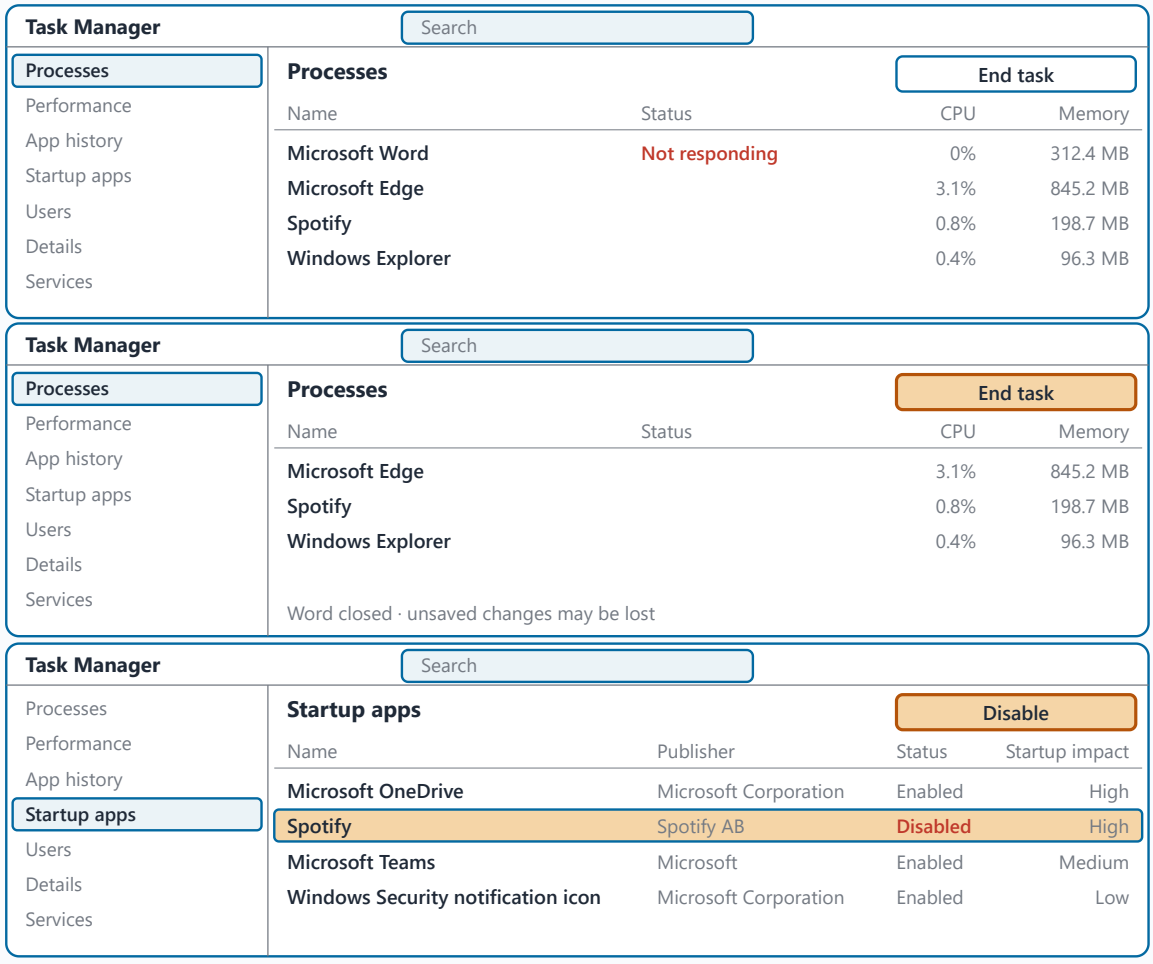


Figure 2.2.31: Task Manager, step by step. At the top, on the Processes page, Word's status is "Not responding." In the middle, Word has been closed with End task. At the bottom, on the Startup apps page, Spotify has been disabled.

Common mistake

Disabling everything in the startup list. Some programs, such as antivirus, a touchpad driver's helper, or cloud sync, need to run at startup. The ones to disable are programs that are not really needed at startup and have a high startup impact: a music player, a game client, a chat app.

2.2.10.5 Updates, backup, and encryption

Windows Update is the service that brings security and quality updates to Windows. On the Settings → Windows Update page, updates are checked for, installed, and their history is viewed. Security updates are usually released on the second Tuesday of each month. Some updates ask for a restart. The *Active hours* setting keeps Windows from making this restart during the hours when the user is working. Updates can be paused for a short time but should not be postponed altogether.

Definition 2.2.27

A **backup** is keeping a copy of important files in a separate location. **Restore** is returning files or the system to an earlier state.

For a backup to be of use, it must not be lost together with the original files. A copy on the same disk goes with it when the disk fails. That is why there is a common rule: the **3-2-1 rule**. Important data should have at least 3 copies, on at least 2 different media, and at least 1 copy should be in another place, for example in the cloud. Figure 2.2.32 shows the Windows counterpart of this scheme: the files on the computer are backed up both to OneDrive and to an external drive.



Figure 2.2.32: Files on the C: drive, encrypted with BitLocker, are backed up to OneDrive (the cloud) and, with File History, to an external drive. The BitLocker recovery key is kept in the account or as a printout, away from the computer.

Definition 2.2.28

BitLocker is the feature that encrypts a whole drive in the supported editions of Windows (Pro, Enterprise, Education). Data on an encrypted drive cannot be read without the right key. The **recovery key** is the 48-digit number that gives access to the drive when the TPM or the password cannot unlock it. In the Windows Home edition, a similar feature exists under the name “Device encryption.”

Encryption protects the data when the computer is stolen. Even if a thief removes the disk and puts it in another computer, the files cannot be read. But the same protection applies to the owner too. If the motherboard is replaced, the TPM is reset, or the UEFI settings change, Windows asks for the

recovery key at startup. If the key is not known, the data can never be accessed again. That is why the recovery key is kept away from the computer: in the Microsoft account, on a printout, or on a USB drive.

Common mistake

Keeping the BitLocker recovery key inside the encrypted drive, for example in a text file in Documents. The key is needed precisely when the drive cannot be opened; at that moment the file inside the drive cannot be reached. The key is always kept outside the computer.

2.2.10.6 Recovery options

The tools to turn to when something goes wrong are ordered from light to heavy by their effect. A light tool brings back only a single file; a heavy tool reinstalls all of Windows. The right choice is the lightest tool that solves the problem (Figure 2.2.33, Table 2.2.24).



Figure 2.2.33: Recovery options from light to heavy: OneDrive backup › File History › Restore point › Reset this PC.

Tool	What does it protect or restore?	Condition and limit
OneDrive folder backup	backs up the Desktop, Documents, and Pictures folders to the cloud; a deleted file returns from the OneDrive recycle bin	requires the internet and a OneDrive account; the cloud bin keeps items for a limited time
File History	regularly saves versions of files to an external drive; an old or deleted version can be restored	the external drive must be connected
Restore point	returns system files and settings to an earlier state	does not back up personal documents; the feature must have been turned on beforehand
Reset this PC	reinstalls Windows; “Keep my files” keeps personal files and removes apps; “Remove everything” deletes it all	a last resort; make a backup first

Table 2.2.24: Recovery tools.

File History is opened from the “File History” page in Control Panel and by default takes a copy every hour. A restore point is enabled from the System Protection tab, which opens when you search for “Create a restore point” in the Start menu. Windows creates a point by itself before major updates and driver installations. Reset this PC is on the Settings → System → Recovery page.

Example 2.2.26

The most suitable recovery tool is chosen for four different problems.

1. **A file deleted yesterday from Documents, which is backed up with OneDrive, is wanted back.** The lightest tool is enough: the file is restored from the OneDrive recycle bin.
2. **Last week's version of a file that is regularly saved to an external disk is wanted.** File History: the external drive is plugged in, and the old version of the file is selected and restored.
3. **After a new device driver update, the system is unstable, but the documents are intact.** The problem is in the system, not in the documents. A restore point returns the system to the point before the update; it does not touch the documents.
4. **The system is completely broken; personal files must be kept, and apps can go.** Reset this PC → Keep my files. Windows is reinstalled, personal files stay, and apps are removed; the ones needed are installed again by hand.

In every case a heavier tool could also have solved the problem, but it would have caused needless losses. For example, resetting the computer for a single deleted file removes all the apps and does not bring the file back either.

Common mistake

Mistaking a restore point for a document backup. A restore point covers only system files, the registry, and installed programs. A deleted homework file does not come back with a restore point. For documents you need OneDrive, File History, or another file backup.

2.2.11 Summary

- The **desktop** is the main work area. The **Start menu** leads to apps, search, the account, and power commands; right-clicking it or pressing Win + X opens the Quick Link menu. The **taskbar** carries open and pinned apps, the system tray, quick settings (Win + A), and notifications (Win + N).
- When a window is **minimized**, the app does not close. **Snap** (Win + arrow keys, Win + Z) arranges windows, and Alt + Tab changes the active window. **Virtual desktops** (Win + Ctrl + D, ← / →, F4) group windows; when a desktop is closed, its windows move.
- Copy (Ctrl + C), Cut (Ctrl + X), and Paste (Ctrl + V) use the **clipboard**. Win + PrtSc saves a file; Win + Shift + S and Alt + PrtSc put the image on the clipboard.
- A physical disk can be divided into several **partitions**; each volume gets a **drive letter**. A 512 GB disk appears in Windows as ≈ 476.8 GiB.

- A file is defined by its name, **extension**, path, and properties. The extension determines the associated app; changing it does not convert the content. Nine characters (\ / : * ? " < > |) and reserved names such as CON and PRN cannot be used; the classic path limit is 260 characters.
- An **absolute path** starts at the root; a **relative path** starts at the current folder, and `..` is the parent folder. The real path of the Documents folder is `C:\Users\<name>\Documents`.
- **Copying** keeps the original and increases the number of files; **moving** changes the location. Drag and drop moves on the same drive and copies across drives; Shift forces a move, Ctrl a copy, and Alt a shortcut.
- Delete sends to the **Recycle Bin**, and Shift + Delete deletes permanently; the bin is not used on USB and network locations. A **shortcut** is a .lnk file that holds only a path; it is not a backup.
- **ZIP** gathers files in one package; text compresses well, while JPEG and DOCX hardly compress at all. A document inside a ZIP is not edited without extracting it. Size on disk is rounded up to 4 KB blocks.
- **Settings** (Win + I) gathers the system, device, network, account, app, language, and accessibility options. If text is too small, change the **scale**, not the resolution.
- A **standard user** is for everyday work; **UAC** asks for confirmation or a password for administrator tasks. A **default app** is a permanent choice, while Open with is a one-time choice.
- A **Windows Hello** PIN is tied to the device and does not pass over the network. **Windows Security** gathers virus protection, the firewall, SmartScreen, and device security.
- Storage Sense and cleanup recommendations free up space. An HDD is **defragmented**, and **TRIM** is applied to an SSD. **Task Manager** ends a non-responding app and limits startup apps.
- Keep the backup in a separate medium (3-2-1). Keep the BitLocker recovery key away from the computer. The recovery tools, from light to heavy: OneDrive, File History, Restore point, Reset this PC.

Calculation	Formula	Example
GB → GiB	bytes ÷ 2^{30}	512 GB → ≈ 476.8 GiB
Size on disk	⌈size ÷ 4,096⌉ × 4,096	1,500 bytes → 4,096 bytes
Compression ratio	compressed ÷ original	31 ÷ 120 ≈ 25.8%
Logical screen area	resolution ÷ scale	1920 × 1080, 150% → 1280 × 720
Usage ratio	used ÷ total	312 ÷ 476 ≈ 65.5%
Single-core load	100 ÷ number of logical processors	8 processors → 12.5%
PIN possibilities	$10^{\text{number of digits}}$	4 digits → 10,000

Table 2.2.25: The main calculations used in the notes.

2.2.12 Key terms

These are the key terms of the chapter, in the order in which they appear.

desktop Start menu taskbar system tray quick settings notification center context menu window title bar minimize / maximize / restore down / close snap task view virtual desktop keyboard shortcut clipboard screenshot Snipping Tool drive partition / volume file folder file name extension file association reserved name path absolute path relative path current folder parent directory root File Explorer navigation pane wildcard hidden drag and drop Recycle Bin permanent delete shortcut (.lnk) archive / compress extract all properties attributes read-only scale user account local account standard user administrator default app startup apps screen reader keyboard layout biometric multi-factor authentication firewall real-time protection Secure Boot Storage Sense defragmentation fragmented TRIM Task Manager process not responding end task backup restore point recovery key reset

---

2.2.13 Test Yourself

1. On the taskbar there is a line under the Word icon but the icon is not highlighted; there is no line under the Excel icon. What is the state of these two apps? What happens when the Excel icon is clicked?

   Answer

   Word is open but is not the active window (it may be in the background or minimized). Excel is only pinned and is not open. When the Excel icon is clicked, Excel starts; a line appears under its icon and Excel becomes the active window. Word stays open.

2. Word is open on Desktop 1 and Music on Desktop 2; the screen shows Desktop 2. Win + Ctrl + D is pressed, then Win + Ctrl + ←. Which desktop are we on, and how many desktops are there? If Win + Ctrl + F4 is then pressed, where does Music go?

   Answer

   Win + Ctrl + D creates a new, empty Desktop 3 and switches to it. Win + Ctrl + ← goes back one to the left, to Desktop 2. There are three desktops, and the screen shows Desktop 2. Win + Ctrl + F4 closes Desktop 2; Music does not close but moves to Desktop 1 on its left.

3. A disk labeled “1 TB” on its box is set up as a single partition, D:. About how many GB does

Windows show as this volume's capacity?

Answer

1 TB = 10^{12} bytes. Windows counts 2^{30} bytes as 1 GB: $10^{12} \div 1{,}073{,}741{,}824 \approx 931.3$. The volume appears as about 931 GB. The number of bytes is the same; the difference comes from the unit.

4. The current folder is `C:\Users\Ali\Pictures\Holiday`. What is the absolute equivalent of the relative path `..\..\Documents\homework.docx`? If the leading `..\..\` had not been written, in which folder would it have been looked for?

Answer

The first `..` goes up from Holiday to Pictures, and the second `..` goes up from Pictures to Ali. Then it goes down into Documents: `C:\Users\Ali\Documents\homework.docx`. Without `..\..\`, the `Documents` folder would have been looked for inside Holiday, that is, as `C:\Users\Ali\Pictures\Holiday\Documents`, and would not have been found.

5. Which of these names cannot be file names in Windows: `note|1.txt`, `PRN.docx`, `printer.docx`, `2026-10-23_report.pdf`, `a<b.txt`?

Answer

`note|1.txt` (| is a forbidden character), `PRN.docx` (PRN is a reserved name; the extension does not change that), and `a<b.txt` (< is a forbidden character) cannot. `printer.docx` and `2026-10-23_report.pdf` are valid.

6. C:\Documents holds `a.docx` and `b.jpg`, and E:\ (a USB drive) is empty. In order: `a.docx` is dragged and dropped onto E:\; `b.jpg` is dragged and dropped onto `C:\Archive`; the `a.docx` on E:\ is dragged to `C:\Archive` while Shift is held. In the end, which files are in which location?

Answer

The first operation is a copy because the drives differ: `a.docx` is in both C:\Documents and E:\. The second is a move because the drive is the same: `b.jpg` goes to C:\Archive. The third is a move because, although the drives differ, Shift is held: the copy on E:\ goes to C:\Archive. Result: C:\Documents holds `a.docx`; C:\Archive holds `b.jpg` and `a.docx`; E:\ is empty.

7. What is the size on disk of a 9,000-byte file on an NTFS volume that uses 4,096-byte blocks? About

how many KB does Windows show as the file's size?

Answer

9,000 ÷ 4,096 ≈ 2.2; rounded up, 3 blocks. The size on disk is 3 × 4,096 = 12,288 bytes (12.0 KB). The size appears as 9,000 ÷ 1,024 ≈ 8.79 KB.

8. A 600 KB text file and a 3,000 KB JPEG photo are added to a ZIP archive. The text compresses to 25% and the photo to 99%. What are the approximate size of the archive and the overall saving (ignore the share of archive information)?

Answer

The text becomes 600 × 0.25 = 150 KB and the photo 3,000 × 0.99 = 2,970 KB. The archive is ≈ 150 + 2,970 = 3,120 KB. The original total is 3,600 KB; the saving is (3,600 − 3,120) ÷ 3,600 ≈ 13.3%. Almost all of the saving comes from the text file.

9. On a screen with a resolution of 2560 × 1600, the scale is set to 200%. What logical area is left for applications? Why is lowering the resolution to 1280 × 800 not a good way to enlarge text?

Answer

2560 ÷ 2 = 1280, 1600 ÷ 2 = 800: the logical area is 1280 × 800. With scale, all the physical pixels are used and the image stays sharp. Lowering the resolution, on the other hand, spreads each pixel over four (2 × 2); the logical area is the same, but the image becomes blurry.

10. A student accidentally deleted a homework file with Shift + Delete; the file was in the Documents folder backed up with OneDrive. Also, after a driver update, the sound card does not work. Which recovery tool suits each problem?

Answer

Shift + Delete does not send the file to Windows' Recycle Bin. But while OneDrive reflects the deletion to the cloud, it keeps the file in its own recycle bin for a while; the file is restored from there. The sound card problem is in the system: if restore points are on, the system is returned to the point before the update. Resetting the computer is needlessly heavy for both problems.

2.2.14 Further reading

- Microsoft Support, *Windows help and learning* (support.microsoft.com/en-us/windows): the official help pages, which describe the desktop, File Explorer, Settings, backup, and recovery with the current menu names of Windows 11.
- Microsoft Support, *Keyboard shortcuts in Windows* (official page): a much longer list of shortcuts than the ones given in these notes, organized by topic.
- Microsoft Learn, *Naming Files, Paths, and Namespaces* (learn.microsoft.com): the technical explanation of forbidden characters, reserved names, absolute and relative paths, and the 260-character limit.
- Microsoft Learn, *Windows Hello for Business overview* (learn.microsoft.com): explains why the PIN does not leave the device and how the private key is protected in the TPM.

2.2.15 References

1. Microsoft Support, “Windows help and learning” documentation (support.microsoft.com/en-us/windows).
2. Microsoft Support, documentation on Storage Sense and drive optimization.
3. Microsoft Learn, Windows Hello and Windows security documentation (learn.microsoft.com).
4. Microsoft Learn, “Naming Files, Paths, and Namespaces,” Win32 file system documentation.

PART I · CHAPTER 3

Office Applications

Office applications make it easier to prepare documents, analyze data and present information. The Word deck covers styles, page layout, section breaks, tables of contents, bibliographies and mail merge. The Excel deck deals with cell references, formulas and functions, sorting and filtering, pivot tables and charts. The PowerPoint deck shows how to plan a presentation, use the slide master, design visuals, add transitions and animations, and deliver an effective talk.

NOTES IN THIS CHAPTER

3.1 Microsoft Word

3.2 Microsoft Excel

3.3 Microsoft PowerPoint

PART I · CHAPTER 3.1

Microsoft Word

Homework reports, lab handouts, petitions, project proposals, a graduation thesis… Most of the documents an engineering student writes are prepared in a word processor. You can use Word like a typewriter: type, make things bold, press Enter. But the approach falls apart as the document grows. Adding one heading means renumbering everything by hand, one page shift leaves the table of contents wrong, and a single wide table turns the whole document to landscape. These notes teach Word the *structure* of a document, not its *appearance*: styles, sections, fields, and automatic lists. The topics proceed in order, and each tool is applied step by step to a small document. Tables track the state the document is in after every step.

LEARNING OUTCOMES

Document management
Create, save, open, and export documents; create a document from a template; explain the difference between Save, Save As, AutoSave, and AutoRecover.

Structural formatting
Tell character-, paragraph-, section-, and document-level formatting apart; build a consistent document by understanding how styles relate to direct formatting.

Long documents
Use heading styles, multilevel numbering, a table of contents, footnotes, captions, and cross-references; manage page orientation, headers, and page numbers per section with section breaks.

Collaboration and sharing
Work together with comments and Track Changes; produce bulk documents with mail merge; check a document for accessibility and hidden content, then print it or share it as a PDF.

3.1.1 Word and the word processor

The simplest way to write text on a computer is a plain-text editor. Notepad in Windows is an example: it stores letters, but it knows nothing about bold type, page numbers, or tables. A report, however, is more than letters. It has headings, paragraphs, tables, figures, footnotes, and a page

layout. A program that keeps and manages all of these elements in a single file is called a word processor.

3.1.1.1 What is a word processor?

Definition 3.1.1

A **word processor** is application software for typing, editing, formatting, laying out, and printing text. A file prepared in a word processor is called a **document**. Microsoft Word is the word processor of the Microsoft 365 suite; by default it stores its documents in files with the `.docx` extension.

What sets a word processor apart from a plain-text editor is that, besides the text, it also stores information about *how the text looks* and *what it means*. That a line is 14 point and blue is information about appearance. That the same line is a "level-1 heading" is information about structure. Word stores both. This distinction is also the most important idea in these notes: if appearance is derived from structure, the document is easy to manage. If appearance is applied by hand to every line, the document falls apart as it grows.

The page Word shows on screen is very close to the page that will come out on paper when printed. This approach is called WYSIWYG (what you see is what you get). Because the screen looks like paper, users tend to adjust formatting by eye. That is the trap. A heading pushed down with empty lines looks right on screen, but it drifts as soon as two lines are inserted above it. For this reason, most of these notes are about building the layout with Word's tools rather than by eye.

3.1.1.2 What Word is used for

The jobs Word is used for fall roughly into six groups. Each group highlights a different capability of Word.

Use	Example	Key tools
Correspondence	petition, letter, official letter	paragraph alignment, templates
Report	lab report, project report, thesis	heading styles, table of contents, tables, captions, bibliography
Forms and templates	application form, meeting minutes	template (`.dotx`), placeholders
Bulk documents	personal invitations or grade notices to hundreds of people	mail merge
Collaboration	a thesis draft corrected by the advisor, a joint project report	comments, Track Changes, co-authoring

Use	Example	Key tools
Publishing	the final version printed on paper or sent by e-mail	print preview, exporting as PDF

Table 3.1.1: What Word is used for, and the tools that stand out in each case.

Table 3.1.1 covers almost all the documents an engineering student will meet over four years. A petition is written in a few minutes and basic tools are enough. A report, by contrast, runs to dozens of pages, is revised several times, and needs the structural tools most. That is why these notes concentrate on long documents such as reports and theses.

3.1.2 The interface

When Word opens, the document area takes up most of the screen. The ribbon holding the commands sits at the top, the status bar at the bottom, and an optional navigation pane on the left. This section introduces the names and roles of these parts. Knowing where the commands are is the foundation for everything else.

3.1.2.1 The workspace

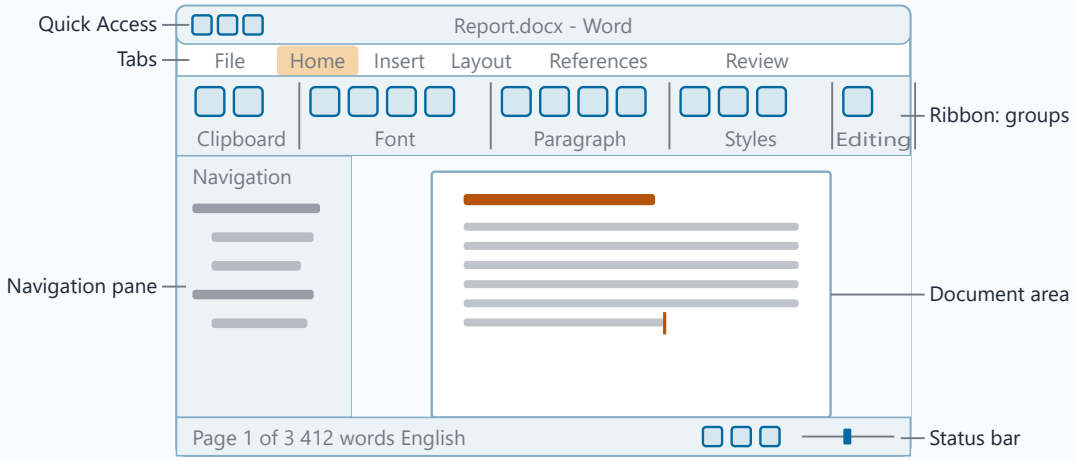


Figure 3.1.1: The parts of the Word window. At the top are the Quick Access Toolbar and the title bar (file name "Report.docx"); below them are the tabs and the ribbon groups (Clipboard, Font, Paragraph, Styles, Editing); on the left is the navigation pane, in the middle the document area, and at the bottom the status bar.

Figure 3.1.1 breaks the Word window into its parts. Each part does the following:

- The **title bar** shows the name of the open document. If the document is saved to the cloud, the AutoSave switch is here too.
- The **Quick Access Toolbar** keeps a few frequently used commands, such as Save and Undo, always visible. The user chooses which buttons stay here.

- The **ribbon** is the region that presents all of Word's commands, divided into tabs and groups.
- The **navigation pane** lists the headings in the document as a tree; clicking a heading jumps to that place. Search results also appear here.
- The **document area** is where the text is typed and the pages are seen. The blinking vertical line, the **cursor**, shows where the next typed character will be inserted.
- The **status bar** shows the page number (“Page 1 of 3”), the word count, the document language, the view buttons, and the zoom slider.

Definition 3.1.2

The **ribbon** organizes commands on three levels: **tab** → **group** → **command**. A tab is one of the names along the top of the ribbon (Home, Insert, Layout…). Each tab is divided into several groups (on the Home tab: Clipboard, Font, Paragraph, Styles, Editing). Each group in turn holds the related command buttons. A **contextual tab** is an extra tab that appears only while a particular object is selected.

In these notes the location of a command is written as “tab → group → command”. For example, “Home → Paragraph → Center” means the Center button in the Paragraph group of the Home tab. The group name is omitted when there is no ambiguity, as in “Layout → Breaks”.

Contextual tabs keep the ribbon uncluttered. When a table is selected in the document, a **Table Design** tab and a separate **Layout** tab for the table appear on the right of the ribbon. When a picture is selected, the **Picture Format** tab appears. When the selection is cleared, these tabs disappear again. A user looking for a table's border color will not find the command on any tab unless they click the table first.

Common mistake

Looking on the ribbon for a table-specific or picture-specific command while no object is selected. These commands live on contextual tabs and are visible only while the object is selected. First click inside or on the object, then switch to the tab that appears on the right of the ribbon.

3.1.2.2 The main tabs

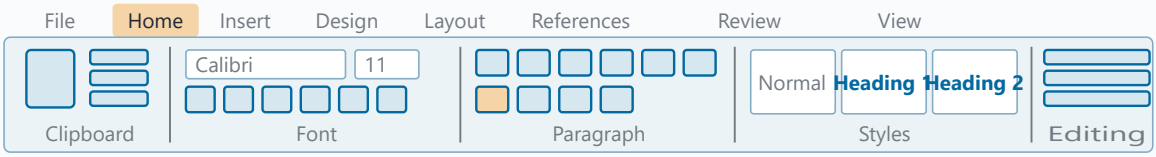


Figure 3.1.2: The ribbon with the Home tab selected. From left to right: the Clipboard, Font (font name "Calibri", size 11), Paragraph, Styles (Normal, Heading 1, Heading 2), and Editing groups.

When Word opens, the **Home** tab is selected, because the most frequently used commands are there (Figure 3.1.2). Each of the other tabs corresponds to a stage of preparing a document. Table 3.1.2 gives this mapping.

Tab	Main role	Example commands
File	operations on the document as a whole	New, Open, Info, Save, Save As, Print, Export
Home	text, paragraph, style, and clipboard operations	Copy, Paste, Bold, Center, Heading 1, Find
Insert	inserting objects into the document	Table, Pictures, Shapes, Link, Bookmark, Header, Page Number
Design	the overall look of the document	Themes, Colors, Fonts, Watermark
Layout	page layout	Margins, Orientation, Size, Columns, Breaks
References	helper elements for a long document	Table of Contents, Insert Footnote, Insert Citation, Insert Caption, Index
Mailings	bulk document production	Envelopes, Labels, Select Recipients, Insert Merge Field
Review	checking and collaboration	Spelling & Grammar, New Comment, Track Changes, Accept
View	how the document is displayed on screen	Read Mode, Print Layout, Navigation Pane, Ruler, Zoom

Table 3.1.2: The main tabs of Word and their roles.

The order of the tabs is not random; it roughly follows the order in which a document is prepared. First the text is typed and formatted (Home), then tables and figures are added (Insert). The page layout is set (Design, Layout), and the table of contents and bibliography are built (References). Last of all, the document is reviewed (Review). The Mailings tab is needed only when producing bulk documents.

Example 3.1.1

For each of five jobs in a report, which tab is the right one?

Job	Tab → command	Rationale
Setting the left and right page margins to 3 cm	Layout → Margins	page layout

Job	Tab → command	Rationale
Adding the text “Figure 1” under a measurement chart	References → Insert Caption	numbered helper element
Accepting the advisor's corrections	Review → Accept	collaboration
Putting the university logo on the cover	Insert → Pictures	inserting an object
Changing the color scheme of the whole document	Design → Colors	document-wide look

On every row the decision rests on the same question: does the job concern the text itself, the page, the document as a whole, or an object added to the document?

3.1.2.3 Document views

The same document can be displayed on screen in different ways. A view changes only the on-screen presentation; it does not touch the content, the formatting, or the printed output of the document. A view is chosen from the View tab or with the small buttons at the right of the status bar.

View	What it shows	When to use it
Print Layout	pages as on paper, with margins and headers	the default; when adjusting the layout
Read Mode	text in columns that fit the screen, without the ribbon	only when reading a long document
Web Layout	text that flows without page breaks, as in a browser	content to be published on the web, wide tables
Outline	headings indented by level; lower levels can be collapsed	when reorganizing the sections of a long document
Draft	text only; no headers, picture positions, or page edges	fast typing and editing

Table 3.1.3: Word's document views.

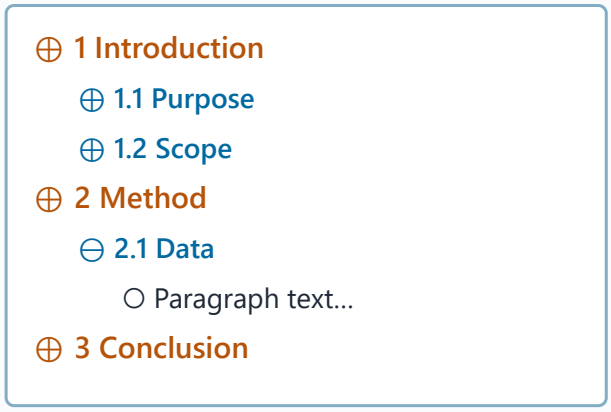


Figure 3.1.3: The Outline view. Headings marked ⊕ have content beneath them and are collapsed; “2.1 Data”, marked ⊖, is expanded and the paragraph (○) under it is visible.

The Outline view is especially useful in long documents (Figure 3.1.3). When a heading is dragged, all the subheadings and paragraphs beneath it move along with it. For example, to move the “3 Conclusion” section ahead of “2 Method”, you drag a single heading in Outline view instead of cutting and pasting dozens of paragraphs. For this view to work, the headings must be written with heading styles. Styles are covered in 3.1.5.5.

3.1.3 Document management

A document's life begins with a blank page or a template. Content is written, saved, reviewed, and finally printed or shared as a PDF. This section covers the file side of that life cycle: the kinds of saving, automatic saving in the cloud, templates, and the internal structure of a document.

3.1.3.1 Creating and saving documents

Preparing a document can be summarized as a five-step flow: open a new document or template, create the content, save, review, and finally export to PDF or print. Saving is not a step done only once in this flow; it is repeated many times during the work.

Definition 3.1.3

Save (`Ctrl+S`) overwrites the same file with the changes in the document open in the window. **Save As** (`F12`) writes the document as a *separate file* with a new name, location, or format; after that, the window works on this new file. When a new document is saved for the first time, Save behaves like Save As and asks for the file name, location, and format.

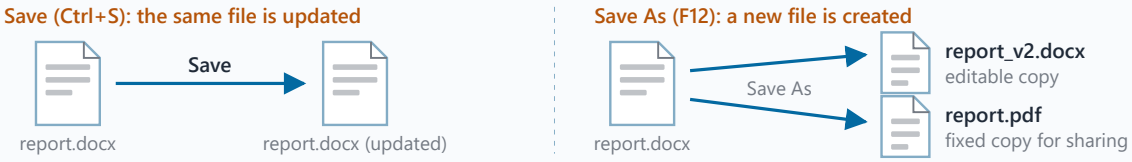


Figure 3.1.4: Save versus Save As. Save updates the same file (`report.docx`). Save As creates a new file: `report_v2.docx` for an editable copy, `report.pdf` for a fixed copy to share.

Figure 3.1.4 shows the difference between the two commands. The little-known consequence of Save As is this: after the command, the window is working not on the old file but on the newly created one. Every later Save updates the new file, and the old one stays as it was. The example below traces this behavior step by step.

Example 3.1.2

The disk holds `experiment.docx`, which contains only the introduction paragraph. The file is opened and the operations below are performed. After each step, what do the files on disk contain?

Step	Action	File open in the window	`experiment.docx`	`experiment_v2.docx`
0	the file is opened	experiment.docx	introduction	none
1	the "Method" paragraph is typed	experiment.docx	introduction	none
2	`Ctrl+S`	experiment.docx	introduction + method	none
3	the "Result" paragraph is typed	experiment.docx	introduction + method	none
4	Save As → `experiment_v2.docx`	experiment_v2.docx	introduction + method	introduction + method + result
5	the "Discussion" paragraph is typed, `Ctrl+S`	experiment_v2.docx	introduction + method	introduction + method + result + discussion

In steps 1 and 3 the change exists only in memory; it reaches the disk only through a save command. In step 4 the "Result" paragraph was never written to the old file. The Save in step 5 writes not to `experiment.docx` but to `experiment_v2.docx`, the file open in the window. The old file stays as it was in step 2.

Which format a document is saved in depends on what will be done with it. A document that will be edited further is kept as a Word document; a final version to be shared without changes is kept as a PDF. Table 3.1.4 summarizes the main file formats Word uses.

Extension	Format	Use
`.docx`	Word document	editable document; the default format
`.docm`	Word document that can contain macros	macro-enabled document; take care when opening it
`.dotx`	Word template	starting file for new documents
`.dotm`	Word template that can contain macros	for example `Normal.dotm`, which holds the defaults for blank documents
`.pdf`	Portable Document Format	a sharing and print copy whose layout stays the same on every device
`.doc`	legacy Word document (2003 and earlier)	for opening old files; not recommended for new documents

Table 3.1.4: The main file formats related to Word.

The last letter of the extension is a clue: `x` marks a file with no macros, `m` a file that can contain macros. Macros are covered in 3.1.11.3. A PDF file can be opened in Word, but Word converts it into an editable document as it opens it, and the layout may break in the conversion. For this reason, the `.docx` original of every document exported to PDF is kept as well.

Common mistake

Keeping an assignment only as a PDF. A PDF preserves the layout but is unsuited to editing; even a small correction requested by the instructor may mean retyping the document. The right habit is to always keep the `.docx` original and to produce the PDF from it for submission.

3.1.3.2 AutoSave, OneDrive, and AutoRecover

Forgetting to save is one of the oldest accidents in document preparation. Microsoft 365 offers two separate safeguards against it. Their names are similar, but they work differently.

Definition 3.1.4

AutoSave is a feature that writes the changes in the document to the file in the cloud automatically every few seconds. It works only on files stored in OneDrive or SharePoint, and it is switched on and off with the toggle in the title bar. **AutoRecover**, by contrast, keeps a separate recovery copy at regular intervals. If Word closes unexpectedly, it offers this copy when it reopens.

The AutoSave toggle cannot be turned on for a local file in a folder on the computer, such as one on the desktop. Such a file is saved by hand with `Ctrl+S`. Pressing `Ctrl+S` does no harm even when AutoSave is on. This habit keeps the document safe without having to think about whether it is local or in the cloud.

A file kept in the cloud has two further benefits. The first is **version history**: older versions of the file are listed with their dates and times. An earlier version can be opened, inspected, and restored if necessary. The second is **co-authoring**: several people can work on the same cloud file at the same time and see each other's changes within moments.

Feature	Where does it write?	When?	Condition
Save	over the original file	whenever the user chooses	works on every file
AutoSave	to the original file in the cloud	continuously, every few seconds	the file must be in OneDrive or SharePoint
AutoRecover	to a separate recovery copy	at regular intervals	helps only after an unexpected shutdown

Feature	Where does it write?	When?	Condition
Version History	older versions in the cloud	kept automatically	the file must be in the cloud

Table 3.1.5: A comparison of the saving-related features.

Example 3.1.3

Two students are working on the same assignment. Elif keeps the file in her OneDrive folder; Can keeps his on the desktop. Neither presses `Ctrl+S` for a whole hour. Then the power goes out.

1. **Elif.** Because the file is in the cloud, AutoSave is on. The changes were written to the cloud every few seconds. When the computer starts again, the file comes back in its latest state; at most the last few seconds are lost.
2. **Can.** AutoSave does not work on a local file. The original file is as it was an hour ago. When Word reopens, it offers the latest AutoRecover copy. That copy contains the changes up to the last recovery moment; anything after that is lost.
3. **Conclusion.** AutoRecover is a safety net, not a substitute for saving. The recovered copy should immediately be saved with Save As as the main file.

The name, position, and even the presence of the AutoSave button can vary with the Word version and its updates. For example, some Office versions sold with a perpetual license may lack this feature. Whether your version has it can be seen in the title bar and in File → Info.

3.1.3.3 Templates

Imagine a student who writes a few petitions in the same format every semester. Each time the margins are set, the date is right-aligned, and a space is left for the signature. The tool that removes this repetition is the template.

Definition 3.1.5

A **template** is a Word file, stored with the `.dotx` extension, that serves as the starting point for new documents. It contains styles, margins, fixed text, and **placeholders** that mark the places to fill in. When a template is opened, Word opens not the template itself but an untitled copy derived from it.

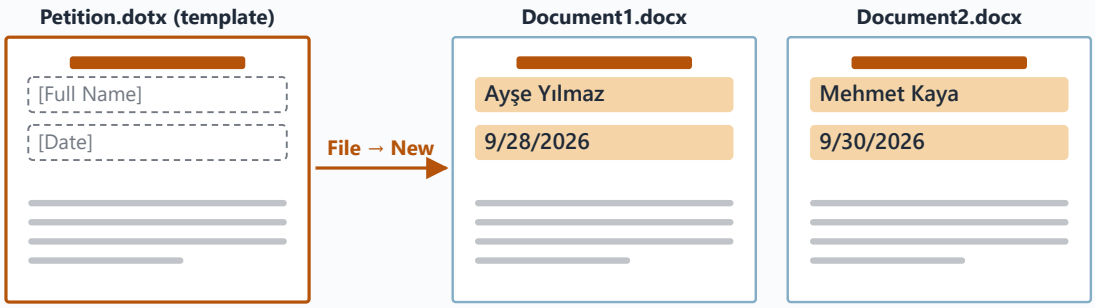


Figure 3.1.5: A new document from a template. The `Petition.dotx` template has the placeholders [Full Name] and [Date]. Each File → New opens a separate document: `Document1.docx` (Ayşe Yılmaz, 9/28/2026) and `Document2.docx` (Mehmet Kaya, 9/30/2026). The template itself does not change.

Figure 3.1.5 shows this behavior. If you double-click the template, fill in the name and date, and then press `Ctrl+S`, Word asks for a file name, because the open document is an untitled copy. The template file remains unchanged, and as many petitions as you like can be produced from the same template.

Templates come from two sources. Ready-made templates are listed on the File → New screen: résumé, report, meeting notes, calendar, and so on. To create your own, first edit a sample document, then choose "Word Template (`*.dotx`)" as the file type in File → Save As.

Word has one more special template: `Normal.dotm`. The default font, size, margins, and styles used when a blank document is opened come from this file. The `m` in the extension shows that this template can also store macros. A change in Normal.dotm affects every blank document opened afterward, so it is touched only deliberately.

Example 3.1.4

A template will be prepared for lab reports. The steps and the result of each step are as follows:

1. In a blank document the margins are set to 2.5 cm, and the Normal and Heading 1 styles are set to the desired font.
2. Fixed text is typed on the cover: the course name, "Experiment No:", "Student:". Placeholders in square brackets mark the places to fill in: [Experiment title], [Full Name], [Date].
3. The headings "Introduction", "Method", "Results", and "Discussion" are written in the Heading 1 style, with the space beneath them left empty.
4. File → Save As → file type "Word Template": `LabReport.dotx`.
5. For each new experiment the template is opened, the placeholders are filled in, and the report is saved under a name such as `Experiment3_Report.docx`. The template waits unchanged for the next experiment.

Common mistake

Opening an old report and deleting its content to write a new one, instead of using a template. Hidden items left over from the old report (comments, pending changes, author information, a forgotten paragraph) are carried into the new document. There is also the risk of accidentally saving over the old file. Preparing a template once for a recurring kind of document is both cleaner and safer.

3.1.3.4 The structural levels of a document

A Word document is not a random heap of letters; it is made of nested levels. Knowing these levels lets you predict in advance which part of the document a formatting command will affect.

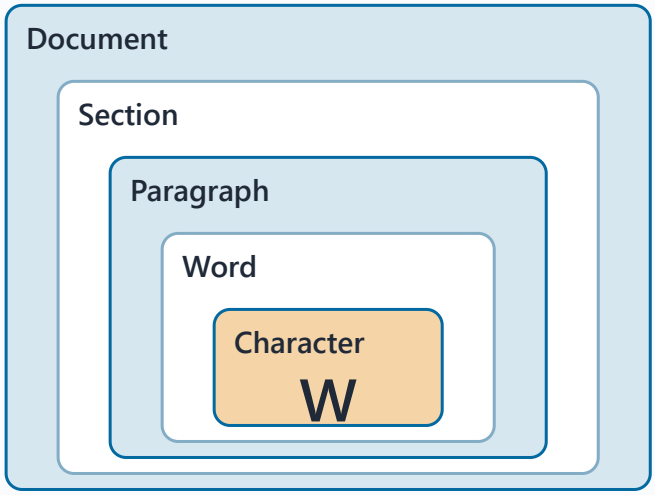


Figure 3.1.6: The structural levels of a document as nested boxes: a document is made of sections, a section of paragraphs, a paragraph of words, and a word of characters.

Definition 3.1.6

The structural levels of a Word document, from the inside out, are: **character** (a single letter, digit, space, or mark), **word**, **paragraph** (the text typed up to the point where Enter is pressed), **section** (a part of the document with its own page settings), and **document**. Every formatting command applies at one of these levels.

Figure 3.1.6 and Table 3.1.6 show these levels together with the formatting set at each. For example, bold is at the character level, alignment at the paragraph level, page orientation at the section level, and the theme at the document level.

Level	Formatting set	How is it applied?
Character	font, size, bold, italic, color, superscript/subscript	to the selected text; with no selection, to the text typed from then on
Paragraph	alignment, indent, line and paragraph spacing, tab stops, border	to the whole paragraph the cursor is in; no selection needed
Section	margins, page orientation, number of columns, header/footer, page-number format	to all the pages of the section the cursor is in
Document	theme, default styles, document properties	to the whole document

Table 3.1.6: The structural levels and the formatting set at each level.

Example 3.1.5

The document has two paragraphs and nothing is selected. With the cursor in the middle of paragraph 2, the Center and Line spacing 1.5 commands are given. Then the cursor is moved to the end of paragraph 1, `Ctrl+B` is pressed, and " End." is typed. What is the result?

1. **Center and line spacing** are paragraph formatting. No selection is needed; the whole of paragraph 2, where the cursor is, is centered and set to 1.5 line spacing. Paragraph 1 is unaffected.
2. **Bold** is character formatting. With no selection, `Ctrl+B` makes bold only the characters typed from then on. " End." is typed in bold; the existing text of paragraph 1 does not change.

Pressing the same key has a different effect depending on the level the command belongs to.

3.1.4 Editing text

Writing a document takes far less time than revising what was written. A student writing a report spends most of the time moving sentences, changing words, and cleaning up unneeded spaces. The tools in this section speed up that work: keyboard movement and selection, the clipboard, undo, and find and replace. Each tool also has a trap, and the traps are shown together with examples.

3.1.4.1 The cursor, movement, and selection

Definition 3.1.7

The **cursor** is the blinking vertical line in the document area; it shows where the next character will be inserted. A **selection** is a highlighted piece of text. Most commands, such as formatting, deleting, cutting, and copying, apply to the selection.

Moving the cursor by clicking with the mouse is enough in short text. In a long document the keyboard is much faster. The arrow keys move the cursor one character or line at a time. The `Ctrl` key enlarges the step: it raises it to the scale of a word, a paragraph, or the whole document. The `Shift` key turns movement into selection: wherever the cursor goes, the text between where it started and where it arrived is selected.

Shortcut	Effect
`Ctrl+Home`	the cursor goes to the very start of the document
`Ctrl+End`	the cursor goes to the very end of the document
`Ctrl+→` / **`Ctrl+←`**	the cursor jumps to the start of the next / previous word
`Ctrl+Shift+→`	makes the same jump and selects the text it passes
`Shift+↑` / **`↓`**	extends the selection one line up / down
`Shift+End`	selects from the cursor to the end of the line
`Ctrl+A`	selects the whole document

Table 3.1.7: Frequently used movement and selection shortcuts.

The mouse can be used for selection too. Double-clicking a word selects the word; clicking three times selects the whole paragraph. A single click in the blank margin area to the left of a line selects that line. To clear a selection, press an arrow key without Shift or click anywhere in the text.

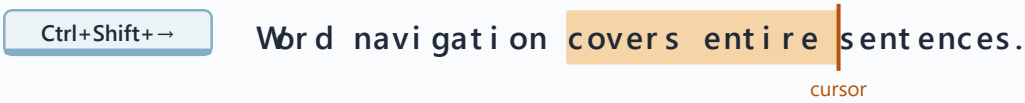


Figure 3.1.7: On the line "Word navigation covers entire sentences.", the state after two `Ctrl+→` presses and two `Ctrl+Shift+→` presses: "covers entire " is selected and the cursor is at the start of the word "sentences.".

Example 3.1.6

The document has a single line: "Word navigation covers entire sentences." The line is 40 characters long. If characters are numbered from 0, the words start at positions 0, 5, 16, 23, and 30. With the cursor in the middle of the line, the keys below are pressed in order. After each step, where is the cursor and what is selected? (Figure 3.1.7 shows the result of the fifth step.)

Step	Key	Cursor position	Selection
1	`Ctrl+Home`	0 (start of "Word")	none
2	`Ctrl+→`	5 (start of "navigation")	none
3	`Ctrl+→`	16 (start of "covers")	none
4	`Ctrl+Shift+→`	23	"covers " (16–22)

Step	Key	Cursor position	Selection
5	`Ctrl+Shift+→`	30 (start of “sentences.”)	“covers entire ” (16–29)
6	`Ctrl+End`	40 (end of document)	none

In steps 4 and 5 the cursor jumps just as in steps 2 and 3; the only difference is that the text it passes is selected. The selection includes the space at the end of a word. If `Ctrl+B` were pressed after step 5, “covers entire” would become bold. The movement without Shift in step 6 clears the selection.

Common mistake

Accidentally pressing a letter key while a selection exists. Word replaces the selected text with the typed character; if the whole paragraph is selected, the paragraph turns into a single letter. There is no need to panic in such a case: `Ctrl+Z` undoes the last operation and the text comes back.

3.1.4.2 Cut, copy, and paste

Definition 3.1.8

The **clipboard** is the memory area where cut or copied content is held temporarily. **Copy** (`Ctrl+C`) puts a copy of the selected content on the clipboard and leaves the source in place. **Cut** (`Ctrl+X`) removes the selected content from the document and puts it on the clipboard. **Paste** (`Ctrl+V`) inserts the clipboard content where the cursor is.

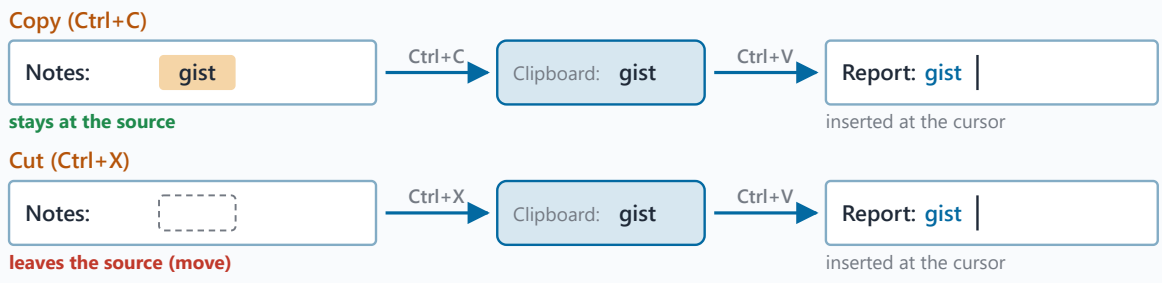


Figure 3.1.8: The difference between Copy and Cut. Top, Copy: the word “gist” stays in the lecture notes, its copy goes to the clipboard and is inserted into the report with `Ctrl+V`. Bottom, Cut: the word leaves the lecture notes; that is, it is moved.

Figure 3.1.8 shows the one difference between the two operations: does the source stay in place? Copy duplicates; Cut moves. Pasting works the same way for both. The clipboard keeps its content after pasting, so the same content can be pasted again as often as you like. The clipboard holds only

the last item cut or copied; a new copy replaces the old one. The Office Clipboard (the small arrow at the bottom-right corner of the Home → Clipboard group), however, can store the last 24 items and lets you paste any one of them.

When text is copied from another source, such as a web page, its color, font, and size come with it. What happens to this formatting on pasting is decided by the **paste options**:

- **Keep Source Formatting**: the text arrives looking as it did at the source. A plain `Ctrl+V` usually does this.
- **Merge Formatting**: the text adapts to the formatting of the place where it is pasted; emphasis such as bold and italic is kept.
- **Keep Text Only**: all formatting is discarded; only the letters arrive and take on the formatting of the destination. In current Microsoft 365 versions its shortcut is `Ctrl+Shift+V`; in older versions the shortcut may differ. In every version it can be chosen from the arrow under the Home → Paste button.

It is also possible to copy formatting separately from the text. The **Format Painter** is the brush button in the Home → Clipboard group. First select the text whose formatting you want to take, and click the brush. Then select the text the formatting is to be applied to. If you double-click the brush, the tool stays on and can be applied in several places in turn; `Esc` turns the tool off.

The power of cut and paste is best seen in a small correction. Consider a three-line recipe. The order of the recipe is wrong: the step “Mix the ingredients” is on the second line but should be first. Instead of deleting the line and retyping it, the line is moved.

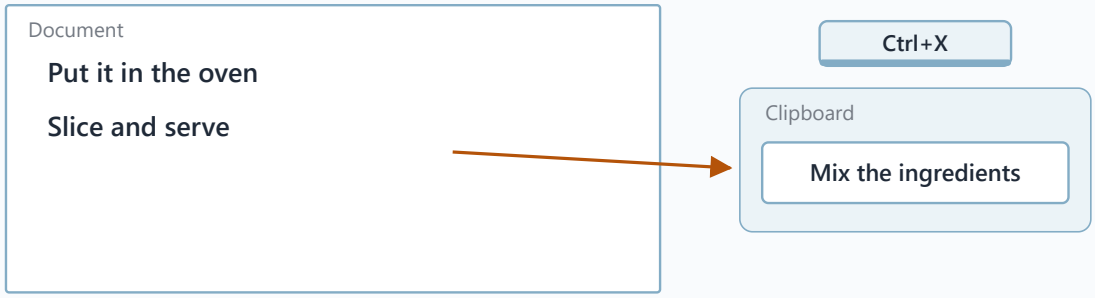


Figure 3.1.9: After `Ctrl+X`: the line “Mix the ingredients” has left the document and been placed on the clipboard. The document is left with “Put it in the oven” and “Slice and serve”.

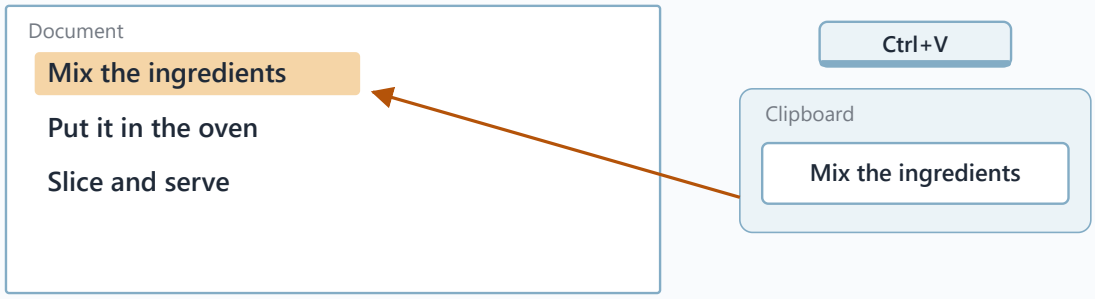


Figure 3.1.10: After `Ctrl+Home` and `Ctrl+V`: the line has been inserted at the start of the document. The clipboard keeps its content.

Example 3.1.7

The steps for moving the line in the recipe, and the state of the document and the clipboard at each step, are as follows (Figure 3.1.9 and Figure 3.1.10):

Step	Action	Lines in the document	Clipboard
0	start	Put it in the oven / Mix the ingredients / Slice and serve	empty
1	line 2 is selected (`Shift+↓` at the start of the line, or a click to the left of the line)	same; line 2 selected	empty
2	`Ctrl+X`	Put it in the oven / Slice and serve	Mix the ingredients
3	`Ctrl+Home`	same; cursor at the start of the document	Mix the ingredients
4	`Ctrl+V`	Mix the ingredients / Put it in the oven / Slice and serve	Mix the ingredients

If Copy (`Ctrl+C`) had been used instead of Cut in step 2, the source line would have stayed in place. In the end “Mix the ingredients” would appear twice in the document: at the top and on the third line. The paragraph mark at the end of the line must also be selected along with the line; otherwise the pasted text merges with the next line.

3.1.4.3 Undo, find, and replace

Word keeps the operations performed in a list, in order. **Undo** (`Ctrl+Z`) reverses the last operation on this list. If pressed repeatedly, the operations are undone one by one, from last to first. **Redo** (`Ctrl+Y`) performs the last undone operation again. When there is nothing to redo, it repeats the last command once more. For example, after one paragraph is made bold, selecting another paragraph and pressing `Ctrl+Y` makes that one bold too.

Definition 3.1.9

Find (`Ctrl+F`) searches the document for places where a text occurs and lists the results in the navigation pane. The **Find and Replace** dialog (`Ctrl+H`) replaces the places where the text typed in the *Find what* box occurs with the text in the *Replace with* box. The **Replace** button replaces the current match, and **Replace All** replaces all matches at once.

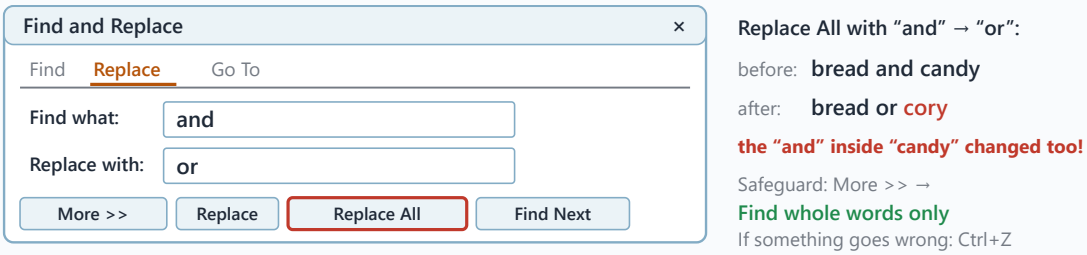


Figure 3.1.11: The Find and Replace dialog. Find what “and”, Replace with “or”. Replace All turns “bread and candy” into “bread or cory”, because the “and” inside the word “candy” is replaced as well. Safeguard: “Find whole words only” among the options opened with the More >> button.

Replace All is a powerful but dangerous command. By default the search looks for a *character string* and pays no attention to word boundaries. Figure 3.1.11 shows the result: searching for “and” also matches the middle of the word “candy”. There are two ways to guard against this trap. The first is to open the search options with the **More >>** button and check the **Find whole words only** box. The second is, instead of Replace All, to first look at the matches one at a time with **Find Next** and press Replace only on the right ones.

Example 3.1.8

The document has a single sentence: “Test results and handwritten notes were collected at home and at school.” Find what: `and`, Replace with: `or`. What is the result of Replace All under the two settings?

1. **Counting the matches.** The string “and” occurs in three places in the sentence, from left to right: the first conjunction “and”, the middle of “handwritten”, and the second conjunction “and”. None of the other words contains it, so there are no other matches.
2. **Default setting.** All three matches are replaced (3 replacements): “Test results or h**or**written notes were collected at home or at school.” The word “handwritten” has been broken.
3. **Find whole words only.** Matches with no letter before or after them are replaced; the match inside “handwritten” is skipped (2 replacements): “Test results or handwritten notes were collected at home or at school.”

If the error is noticed, a single `Ctrl+Z` undoes all the replacements at once, because Replace All counts as one operation.

Find and Replace can look for not only letters but also special items in the document: paragraph marks, tabs, page breaks, and formats. For that you need to know the hidden marks.

Common mistake

Replacing a short word with Replace All without previewing. Changing “cat” to “dog” turns “catalog” into “dogalog” and “category” into “dogegory”. For short strings that can occur inside other words, either check “Find whole words only” or check the matches one by one with Find Next.

3.1.4.4 Hidden formatting marks

A Word document contains characters that are not visible on screen but carry the structure of the document. Every Enter inserts a paragraph mark, every space a space character, every Tab a tab character. These characters are not printed. But most errors in a document come from them: stacked empty paragraphs, double spaces, alignment done with spaces.

Definition 3.1.10

Formatting marks are the on-screen symbols for the non-printing characters in a document. They are switched on and off with the button (Show/Hide) in the Home → Paragraph group. When on, every paragraph end appears as , every space as a middle dot (·), and every tab as an arrow (→).

Mark	Meaning	How does it arise?	Find and Replace code
¶	paragraph mark	`Enter`	`^p`
↵	line break; the same paragraph continues	`Shift+Enter`	`^l`
·	space	space bar	(space)
→	tab	`Tab`	`^t`
°	non-breaking space: the words on either side never fall on separate lines	`Ctrl+Shift+Space`	`^s`
····Page Break····	manually inserted page break	`Ctrl+Enter`	`^m`

Table 3.1.8: Hidden formatting marks, how they arise, and their codes in Find and Replace.

The most important of the marks in Table 3.1.8 is . All of a paragraph's formatting information, such as alignment, indent, and spacing, is stored in this mark. If the at the end of a paragraph is deleted, the paragraph merges with the next one. The merged paragraph takes the formatting stored in the second paragraph's . That is why deleting a can suddenly change the alignment of the text.

The difference between a line break (↵) and a paragraph mark ( ) matters most in headings. If a long heading needs to be split over two lines, `Shift+Enter` is used. The heading then remains a single paragraph and appears as one line in the table of contents. Had `Enter` been used, the heading would have been split into two separate paragraphs and listed as two separate headings in the table of contents.

The non-breaking space is a small but useful tool. Splitting expressions such as “10 kg”, “Figure 3”, or “Dr. Kaya” across two lines at a line end makes reading harder. If the space between the parts is typed with `Ctrl+Shift+Space`, the expression always stays on the same line.

3.1.4.5 Cleaning up a document with Find and Replace

When hidden marks are combined with Find and Replace, cleanup that would take hours by hand comes down to a few clicks. Consider a short report as an example. In the report, the space between paragraphs is made with empty paragraphs, and there are double spaces in two places.

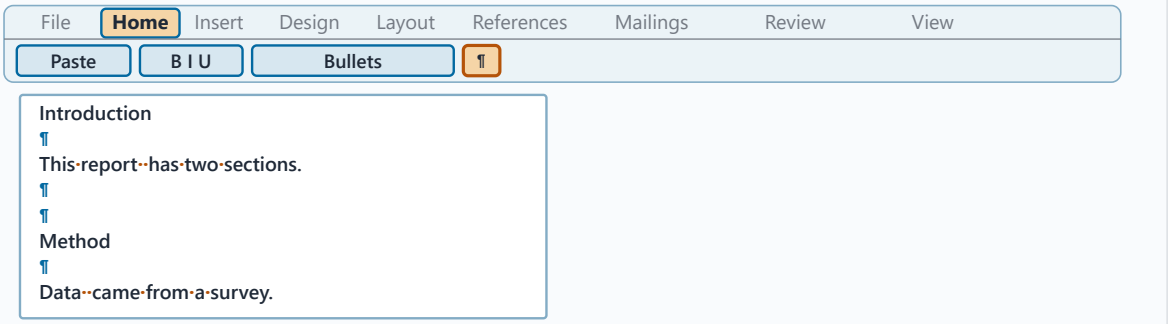


Figure 3.1.12: The document with the button on. There is one empty paragraph between “Introduction” and “This report…”, two between “sections.” and “Method”, and one between “Method” and “Data…”. Double spaces are visible between “report··has” and “Data··came”.

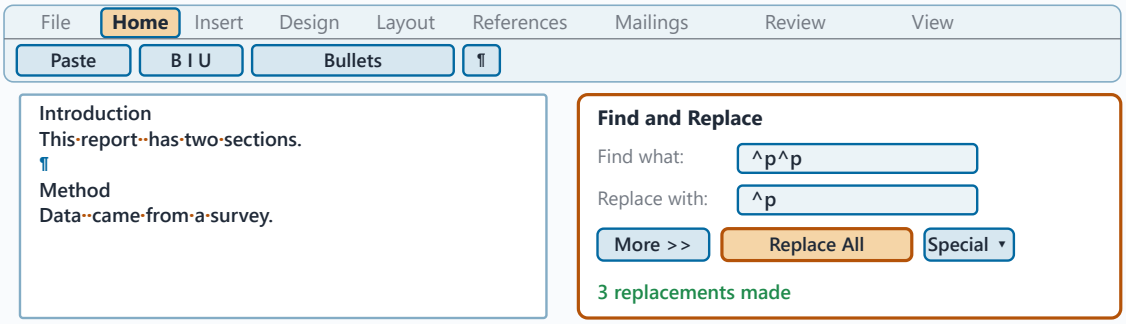


Figure 3.1.13: After the first Replace All with Find what `^p^p` and Replace with `^p` : 3 replacements made. One empty paragraph still remains after the “sections.” paragraph.

An empty paragraph means two paragraph marks in a row: the of the previous paragraph and the empty paragraph's own . So to delete empty paragraphs, `^p^p` is typed in the Find what box and `^p` in the Replace with box. The codes can be typed by hand, or inserted by choosing “Paragraph Mark” from the More >> → Special menu.

Example 3.1.9

With its hidden marks, the document is a character sequence like this (paragraph end, · space):

```
Introduction This·report··has·two·sections. Method Data··came·from·a·survey.
```

The cleanup takes four commands. After each command the state of the document is as follows (Figure 3.1.12 and Figure 3.1.13):

Command	Find what → Replace with	Replacements	Document
1. Replace All	`^p^p` → `^p`	3	Introduction This…sections. Method Data… survey.
2. Replace All	`^p^p` → `^p`	1	Introduction This…sections. Method Data… survey.
3. Replace All	`^p^p` → `^p`	0	unchanged
4. Replace All	two spaces → one space	2	This·report·has…, Data·came…

Why didn't the first command clean everything? The search moves left to right and does not go back to the place it has just replaced. After “sections.” there are three in a row. Word finds the first two and reduces them to a single , then continues the search from the third . The newly written and the third now stand side by side and form a pair. But because the search

does not go back, it does not see this pair; after the third comes “Method”. So two remain at this point, that is, one empty paragraph. The second command catches this pair. The third command finding 0 results shows the job is done.

The general rule is this: if there are *k* empty paragraphs in a row, each pass removes about half of them. That is why the command is repeated until the result is 0. After cleanup, the space between paragraphs should be made not with empty Enters but with the paragraph's **After** spacing (3.1.5.3).

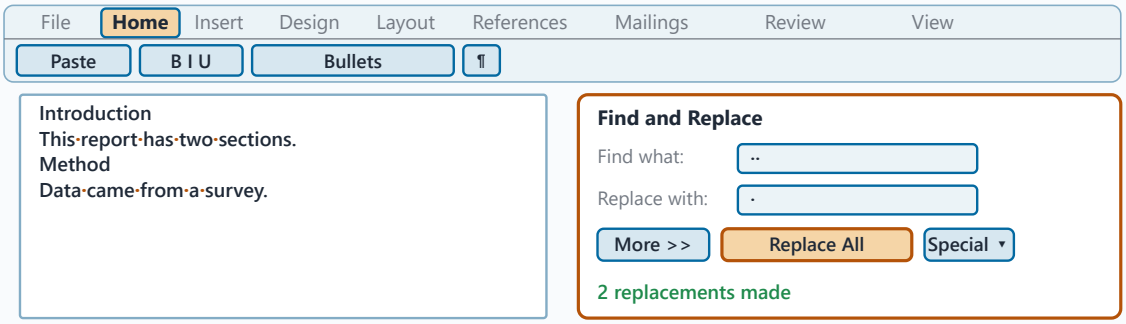


Figure 3.1.14: The final state: no empty paragraphs or double spaces remain (two spaces → one space, 2 replacements).

3.1.4.6 Shortcuts, bookmarks, and hyperlinks

The keyboard shortcuts for frequent tasks turn into a habit of the hand over time. Table 3.1.9 collects a few important, frequently used shortcuts in one place.

Shortcut	Action
`Ctrl+Enter`	inserts a page break
`Ctrl+Shift+V`	pastes text only (in current Microsoft 365 versions)
`Ctrl+B` / `I` / `U`	bold / italic / underline
`Ctrl+K`	inserts a hyperlink
`Ctrl+S` / `F12`	save / save as
`F9`	updates the selected fields

Table 3.1.9: Frequently used shortcuts.

Definition 3.1.11

A **bookmark** is a name given to a point or a piece of text in the document; it is added with Insert → Bookmark, and its name cannot contain spaces. A **hyperlink** (`Ctrl+K`) is a link that takes you to a destination when clicked. The destination can be a web address, another file, or a heading or bookmark in the document.

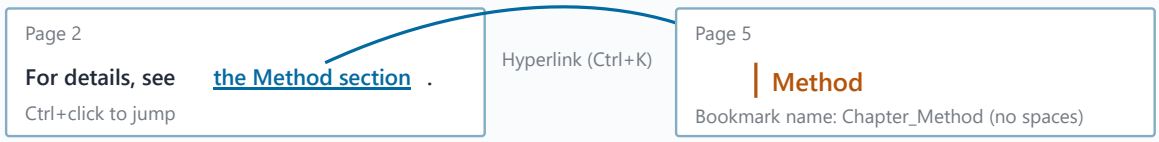


Figure 3.1.15: An in-document hyperlink. The link "the Method section" on page 2 is tied to the bookmark named `Chapter_Method` on page 5; `Ctrl`+click jumps there.

Figure 3.1.15 shows how an in-document hyperlink works. In Word, following a hyperlink by default requires holding `Ctrl` while clicking. A plain click puts the cursor in the link text so that it can be edited. A bookmark is used to make a point that is not a heading, such as the middle of a table, into a destination. If the destination is a heading, no bookmark is needed; the hyperlink can point to the heading directly.

The **View** tab should also be mentioned here. Read Mode, the Navigation Pane, the Ruler, and Zoom are opened from there. The navigation pane lets you jump between headings in a long document. The ruler makes it easier to see paragraph indents and to adjust them by dragging.

3.1.5 Formatting

Formatting is the whole set of settings that determine the appearance of the text. The levels in 3.1.3.4 apply here too: some settings apply to the character, others to the paragraph. In a short document, giving formats one by one causes no trouble. In a long document, repeating the same settings dozens of times takes time and creates inconsistency. This section introduces character and paragraph formats, then the tool that removes this repetition, the style.

3.1.5.1 Character formatting

Definition 3.1.12

Character formatting is the set of settings that determine the appearance of individual characters: the **font** (the visual family of the characters, such as Calibri or Times New Roman), the **size** (in points), emphasis (bold, italic, underline, color), case conversion, **superscript**, and **subscript**. These settings are in the Home → Font group and apply to the selected text.

Text size is measured in **points** (pt). The point is a unit of length that comes from printing: 1 inch is 72 points, and 1 inch is 2.54 cm. From this the millimeter equivalent of a point can be found.

Example 3.1.10

How big is 12 points?

1. 1 pt = 1/72 in = 25.4 mm ÷ 72 ≈ 0.353 mm.
2. 12 pt = 12 × 0.353 mm ≈ 4.23 mm.
3. For comparison: 72 pt is exactly 1 inch, that is, 2.54 cm.

The point size is not the height of the letter itself but the height of the font's design box. This box extends from the top of the letter that rises highest (b, l) to the bottom of the letter that descends lowest (g, p). That is why a 12-point "a" is shorter than 4.23 mm. Two different fonts at the same point size can also look different in size.

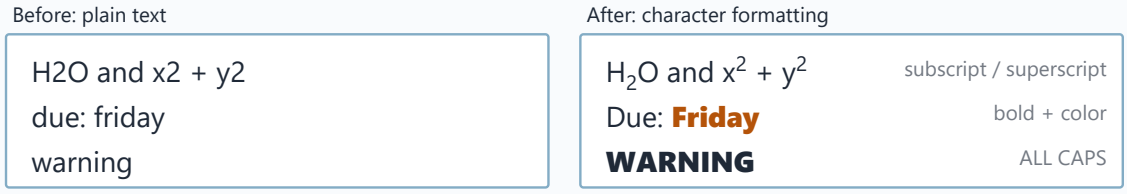


Figure 3.1.16: Character formatting before and after. "H2O" becomes H_2O with subscript and "x2 + y2" becomes $x^2 + y^2$ with superscript; "Due: Friday" is made bold and colored; "warning" is converted to capitals.

Figure 3.1.16 shows formats that are common in engineering texts. **Subscript** is used in chemical formulas (H_2O, CO_2), **superscript** in exponents and units (x^2, m^3, 10^{-6}). These formats are not merely appearance: "m3" and "m^3" mean different things. The **Change Case** command is the Aa button in the Home → Font group. Without retyping the selected text, it converts it to UPPERCASE, lowercase, or Capitalize Each Word.

The **Clear All Formatting** command (the A button with an eraser in the Home → Font group) removes all manually applied character formats from the selected text and returns the text to the default look of its style. It is the fastest way to fix text that looks odd after being pasted from another source.

Common mistake

Selecting the headings of a long document one by one and changing the font, size, and color by hand. If one of twenty headings is forgotten, or one is given 16 points instead of 15, the document becomes inconsistent. If the look changes later, all twenty headings have to be fixed again. Styles are used for repeated formats (3.1.5.5).

3.1.5.2 Paragraph formatting

Definition 3.1.13

A **paragraph** in Word is the text between two paragraph marks ( ). **Paragraph formatting** is the set of settings applied to a paragraph as a whole: **alignment**, **indent**, **line spacing**, **paragraph spacing**, tab stops, border, and shading. These settings are in the Home → Paragraph group and are stored in the paragraph's mark.

No selection is needed for paragraph formatting; if the cursor is anywhere in the paragraph, the command applies to the whole paragraph. To format several paragraphs, select at least a piece of each.



Figure 3.1.17: The four alignment types: left, center, right, and justified. In justified text both ends of the lines reach the margins; the last line stays left-aligned.

Figure 3.1.17 compares the alignment types. Body text is usually **left-aligned** or **justified**. Headings and the cover page are **centered**, and the date in a petition is **right-aligned**. In justified text Word widens the gaps between words to make a line reach the edge. In narrow columns or with long words these gaps can be conspicuous; in that case hyphenation (3.1.6.1) can be turned on.

Tabs are used to set text at particular horizontal positions on the page. For example, if in a petition you want all the values after “Full Name:” to start at the same alignment, you use the Tab key and a tab stop. **Borders and shading** set a paragraph apart from its surroundings with a frame or a background color; they are suitable for making a warning note stand out.

Common mistake

Aligning text with the space bar. A line such as “Full Name: Ayşe” that looks right on screen shifts when the font or size changes, because the width of space characters depends on the font. Use a tab, an indent, or a table without borders for alignment.

3.1.5.3 Indent and spacing measurements

An indent determines how far in from the margin a paragraph starts. Spacing is the vertical space left between lines and between paragraphs. These four measurements have the greatest share in whether a document looks “readable” or “cramped”.

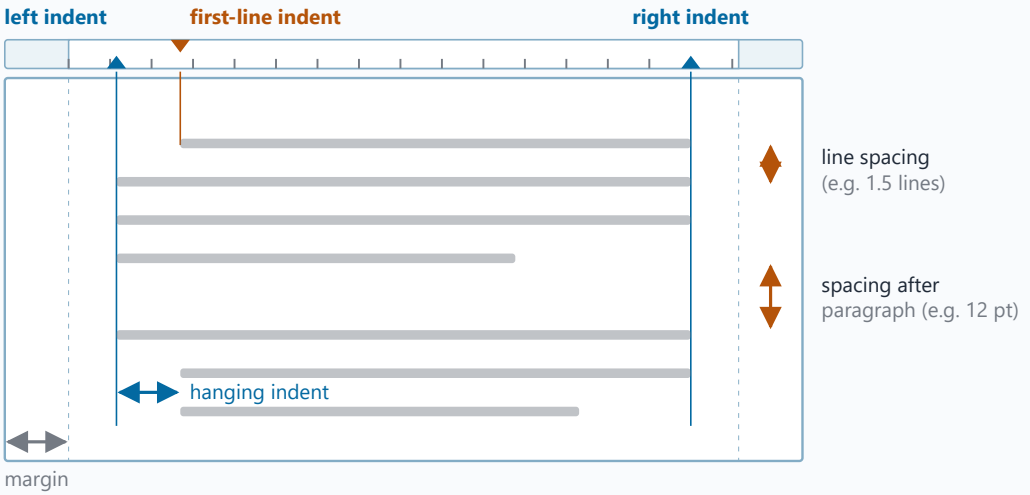


Figure 3.1.18: Paragraph measurements. The ruler has three markers: the upper triangle sets the first-line indent, the lower triangle sets the left (hanging) indent, and the triangle on the right sets the right indent. Paragraph 1 has a first-line indent; paragraph 2 has a hanging indent, as in a bibliography; line spacing (e.g. 1.5 lines) and spacing after the paragraph (e.g. 12 pt) are also shown.

Definition 3.1.14

The **left and right indents** are how far in from the left and right margins all the lines of a paragraph stand. The **first-line indent** pushes in only the first line of a paragraph. The **hanging indent** is its opposite: the first line stays at the left and the following lines are indented. **Line spacing** is the distance between the lines of a paragraph. **Paragraph spacing** is the space left before (Before) and after (After) a paragraph; in Word it is given in points (pt).

Figure 3.1.18 shows these measurements on the ruler. The markers on the ruler can be adjusted by dragging, but for an exact value click the arrow at the bottom-right corner of the Home → Paragraph group. In the Paragraph dialog that opens, indents are typed in centimeters and spacing in points. The “pt” unit in Word is the same as the point used for font size: 1 pt ≈ 0.353 mm.

The first-line indent is the traditional way of marking the start of a paragraph in running text; a common value is 1.25 cm. The hanging indent is used especially in a bibliography. The first line of each source starts at the left and the continuation lines stay indented. That way the reader easily tells where each source begins.

Example 3.1.11

In a report set in 12-point text, a spacing of “After: 12 pt” will be used between paragraphs instead of an empty line. How many millimeters is this space? If the line spacing is set to “Exactly 18 pt”, how many millimeters does one line take up?

1. **Spacing after a paragraph.** 12 pt × 0.353 mm ≈ 4.23 mm. This is the same height as the font box of an empty 12-point line; the look is close to an empty line, but there is no extra in the document.
2. **Line height.** “Exactly 18 pt” gives each line 18 × 25.4 ÷ 72 = 6.35 mm. This value is 1.5 times 12 points.

The “1.5 lines” option in the line-spacing list, on the other hand, is calculated from the font. Word multiplies the single-line height that the font itself suggests by 1.5; for most fonts this height is slightly larger than the point size. That is why “1.5 lines” and “Exactly 18 pt” are close but usually not equal.

Common mistake

Pressing Enter twice to leave space between paragraphs. When empty paragraphs land at the end of a page, they leave a gap at the top of the new page, the layout is disturbed, and cleanup is needed as in 3.1.4.5. Paragraph spacing, on the other hand, shows only between paragraphs and moves with the paragraph.

3.1.5.4 Lists

Reports present a lot of information as lists: lab materials, procedure steps, section headings. Word offers three types of list, and all are in the Home → Paragraph group.

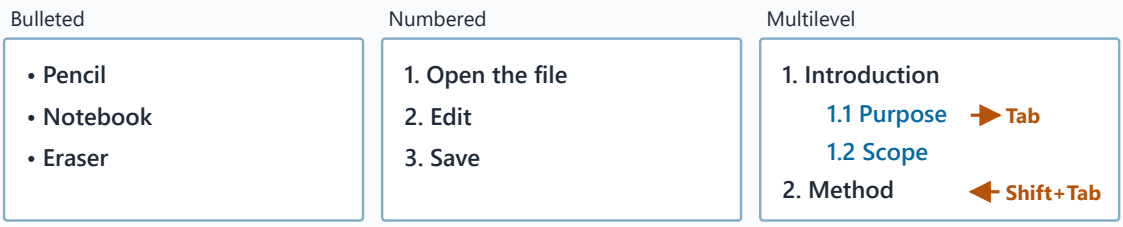


Figure 3.1.19: List types. A bulleted list (Pencil, Notebook, Eraser), a numbered list (Open the file, Edit, Save), and a multilevel list (1. Introduction, 1.1 Purpose, 1.2 Scope, 2. Method). `Tab` demotes an item one level, `Shift+Tab` promotes it one level.

- A **bulleted list** is for items whose order does not matter, such as lab materials.
- A **numbered list** is for items whose order matters: procedure steps, order of priority.

- A **multilevel list** builds a hierarchy of main items and sub-items: 1, 1.1, 1.1.1, and so on.

The level-changing rule in Figure 3.1.19 is the most useful feature of lists. If `Tab` is pressed at the start of a list item, the item drops to a lower level; its number also changes from "3." to a lower-level number such as "a." or "2.1". `Shift+Tab` raises the item one level. Pressing Enter on an empty item at the end of the list ends the list.

Example 3.1.12

A numbered list has three items: 1. Introduction, 2. Method, 3. Conclusion. Enter is pressed at the end of "Method", "Data collection" is typed on the new line, and the operations below are performed in order:

Action	List
Enter, "Data collection" is typed	1. Introduction · 2. Method · 3. Data collection · 4. Conclusion
`Tab` at the start of "Data collection"	1. Introduction · 2. Method · a. Data collection · 3. Conclusion
`Shift+Tab` on the same line	1. Introduction · 2. Method · 3. Data collection · 4. Conclusion

At every step the number of the "Conclusion" item updates by itself. Had the numbers been typed by hand, every insertion and level change would have meant correcting all the numbers one by one. Whether a lower level appears as "a." or "2.1" depends on the chosen list format.

Common mistake

Typing list marks by hand: typing "1." or "-" at the start of a line and continuing with a space. Word sometimes turns this into an automatic list and sometimes does not. In a hand-typed list the indents stay inconsistent, and the numbers shift when an item is inserted. Start a list with the list buttons on the ribbon.

3.1.5.5 Styles and themes

The character and paragraph formats described so far can be applied one by one. But in a report the same look is repeated over and over: all the main headings are formatted the same way, and so are all the body paragraphs. The tool that gathers these repeated formats under a name is the style.

Definition 3.1.15

A **style** is a named package of formatting: it defines settings such as font, size, color, alignment, indent, and spacing together under a single name. Among Word's built-in styles are **Normal** (body text), **Heading 1** (main section heading), and **Heading 2** (subsection heading). Styles are applied from the Home → Styles gallery. A **theme** is the document's overall system of colors, fonts, and effects; it is chosen from the Design tab.

The power of a style comes from a single definition. When Heading 1 is applied to a heading, that heading takes the look in the style's definition. If the style's definition is changed later, all the headings that use this style are updated together. Instead of fixing twenty headings one by one, a single definition is changed.



Figure 3.1.20: Three headings that use the Heading 1 style (1 Introduction, 2 Method, 3 Conclusion). Heading styles also build the document hierarchy: the same headings are listed in the navigation pane.

The second job of heading styles is more important than appearance: they build the structure of the document (Figure 3.1.20). Word recognizes paragraphs in the Heading 1, Heading 2, and Heading 3 styles as “headings”. The navigation pane, the table of contents, the Outline view, and the bookmarks in a PDF are all generated from this information. A line that is merely typed large and bold is, to Word, an ordinary paragraph, however much it looks like a heading.

The relationship between theme and style is layered. The theme determines the document's color palette and its two basic fonts (for headings and for body text). Styles use this palette and these fonts. When another theme is chosen from the Design tab, the styles keep their definitions but take their colors and fonts from the new theme. For example, if Heading 1 is defined with “theme color 1”, all Heading 1s switch to the new theme's first color when the theme changes. A style change, by contrast, affects only the items that use that style.

Common mistake

Leaving a heading in the Normal style and simply typing it bold and large. The look is right, but the navigation pane stays empty and an automatic table of contents does not see this heading. Headings are always written with heading styles; if the look is not liked, the style's

definition is changed.

3.1.5.6 Style definitions and direct formatting

Text with a style applied can also be given formatting from the ribbon. For example, a heading in the Heading 1 style can be made red with the ribbon's color button. In that case the look of the heading comes from two sources: the style definition and the manually applied format. Knowing which one wins prevents most of the surprises that come up when working with styles.

Definition 3.1.16

Direct formatting is formatting given to text one item at a time, from the ribbon or with a shortcut, rather than through a style. Direct formatting takes precedence over the style: even if the style's definition changes, the manually applied property is kept on that text. `Ctrl+Space` removes the manually applied character formats from the selected text and ties the text back to the style's definition.

A style definition is changed in two ways. Right-clicking the name of a style in the Styles gallery opens a menu. The **Modify…** option lets you edit the style's properties in a dialog. **Update Heading 1 to Match Selection**, on the other hand, takes the definition from the look of the currently selected text. The second way is handy for first trying out a heading by hand and then spreading the look you like to all headings.

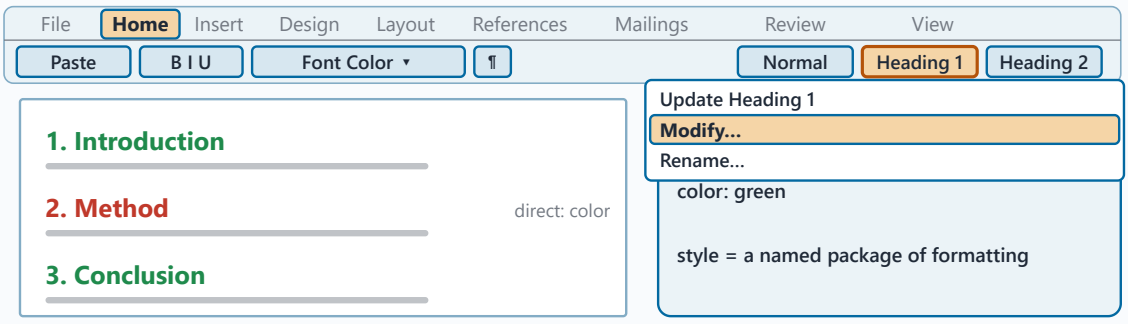


Figure 3.1.21: With Modify…, the definition of Heading 1 was set to 16 pt and green. All three headings changed size; but heading 2 (“Method”), which had earlier been made red by hand, stayed red.

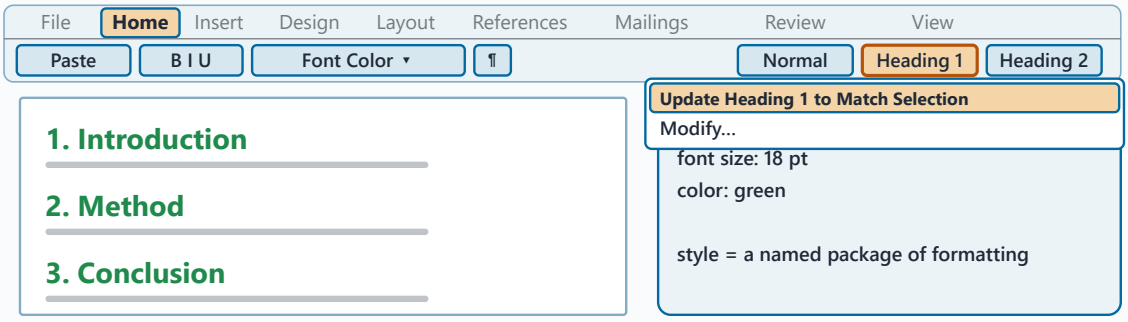


Figure 3.1.22: After heading 3 was set to 18 pt by hand, "Update Heading 1 to Match Selection" was chosen. The definition was taken from heading 3: all Heading 1s became 18 pt.

Example 3.1.13

Three headings (1. Introduction, 2. Method, 3. Conclusion) are in the Heading 1 style. After each step, the style's definition and the look of the three headings are as follows (Figure 3.1.21 and Figure 3.1.22):

Step	Action	Heading 1 definition	Introduction	Method	Conclusion
1	Heading 1 is applied to the three headings	14 pt, blue	14, blue	14, blue	14, blue
2	Method is made red from the ribbon (direct)	14 pt, blue	14, blue	14, **red**	14, blue
3	With Modify..., the definition is set to 16 pt, green	16 pt, green	16, green	16, **red**	16, green
4	Method is selected and `Ctrl+Space` is pressed	16 pt, green	16, green	16, green	16, green
5	Conclusion is set to 18 pt by hand (direct)	16 pt, green	16, green	16, green	**18**, green
6	With Conclusion selected, Update to Match Selection	18 pt, green	18, green	18, green	18, green

Step 3 is the rule itself: when the definition changed, the size of all the headings changed, because the size had not been given by hand to any heading. The color of Method, on the other hand, did not take the style's new color because it had been given by hand. In step 4, once the manually applied color was removed, the heading was tied to the style again. In step 6 the definition was taken from one heading and the new definition spread to all three headings at once.

This example shows the basic habit of working with styles: the look is changed through the style's definition, not by hand. Formats applied by hand gradually build up "hidden exceptions" in a document. When you meet a heading whose look will not change, the first place to look is a forgotten

direct format on that heading. The Word Styles, Navigation Pane and Table of Contents simulator lets you try this interaction.

3.1.6 Page layout

Character and paragraph formats determine the look of the text. Page layout defines the paper the text sits on: the size of the paper, its orientation, margins, number of columns, header, and footer. All of these settings are at the **section** level. If a document consists of a single section, all pages have the same layout. When different pages need different layouts, the document is divided into parts with section breaks.

3.1.6.1 Page setup

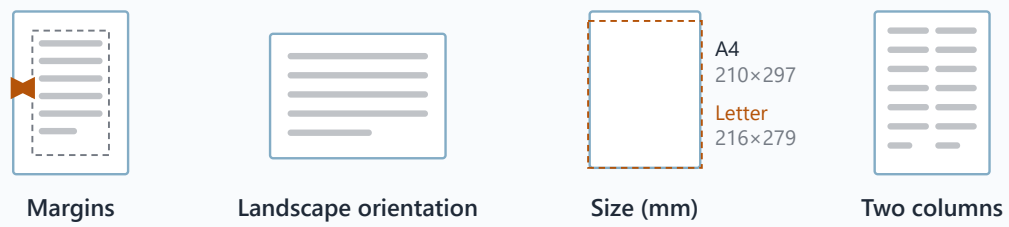


Figure 3.1.23: Page layout options: margins, landscape orientation, a size comparison of A4 (210 × 297 mm) and Letter (216 × 279 mm), and a two-column layout.

Page settings are collected in the Page Setup group on the Layout tab (Figure 3.1.23):

- **Margins**: the blank area between the text and the edge of the paper. For documents that will be bound, an extra **gutter** can be added on the binding side.
- **Orientation**: the page is **portrait** or **landscape**.
- **Size**: the paper size. A4 (210 × 297 mm) is common in Turkey and Europe, Letter (8.5 × 11 in ≈ 216 × 279 mm) in the USA.
- **Columns**: divides the text into columns similar to a newspaper layout. When the text reaches the end of the first column, it continues at the top of the second.
- **Line Numbers**: numbers the lines in some legal or refereed documents, so that a correction can be pointed to as “line 12”.
- **Hyphenation**: splits long words at the end of a line with a hyphen, reducing large gaps in justified text.

Example 3.1.14

On an A4 page, all four margins are 2.5 cm. How much space is left for text? With line spacing "Exactly 18 pt", what is the maximum number of lines that fit on a page?

1. **Width.** 210 mm − 2 × 25 mm = 160 mm.
2. **Height.** 297 mm − 2 × 25 mm = 247 mm.
3. **Line height.** 18 pt = 18 × 25.4 ÷ 72 mm = 6.35 mm (3.1.5.3).
4. **Number of lines.** 247 ÷ 6.35 ≈ 38.9. Half a line does not fit on the page; round down: 38 lines.

If the same document is printed on Letter size, the page is 6 mm wider but 18 mm shorter. The number of lines decreases and the page breaks shift. For that reason the paper size of a document is set before you start writing, according to where it will be submitted.

3.1.6.2 Page breaks and section breaks

There are wrong and right ways to start the next content on a new page. The wrong way is to press Enter until the page ends. The right way is to insert a **break**. Breaks are in the Layout → Breaks menu and fall into two main types.

Definition 3.1.17

A **page break** (`Ctrl+Enter`) starts the following content on a new page; the document stays in the same section. A **section break** divides the document into two sections. Each section can have its own margins, page orientation, number of columns, header, footer, and page-number format.

Break type	What does it do?	When to use it
Page break	new page; same section	starting a new chapter heading at the top of a page
Next Page section break	new section and new page	when the new page will have a different layout (landscape page, different number format)
Continuous section break	new section, on the same page	switching from one column to two columns on the same page
Odd Page / Even Page section break	starts the new section on the next odd- or even-numbered page	always starting chapters on the right-hand (odd) page in books and theses

Table 3.1.10: Break types and where they are used.

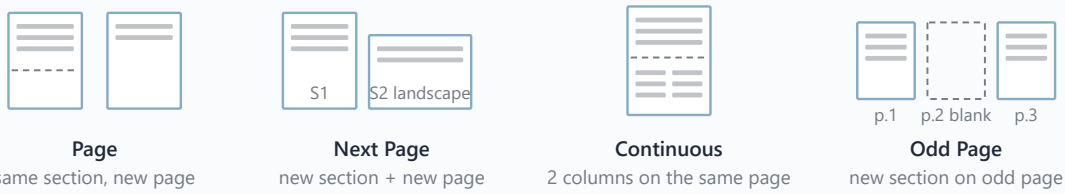


Figure 3.1.24: The effect of the break types. Page break: a new page in the same section. Next Page: the new section (S2) starts on a new page and can be landscape. Continuous: a switch to two columns on the same page. Odd Page: the new section starts on an odd-numbered page; if necessary, an empty even page (p.2) is inserted.

Figure 3.1.24 shows the four breaks side by side. With the button on, breaks appear as dashed lines such as “Page Break” or “Section Break (Next Page)”. If a break is in the wrong place, this line is selected and deleted. A section break has a subtlety: the section's layout settings are stored in the section-break mark at the *end* of the section. When a section break is deleted, the two sections merge, and the merged section takes the settings of the following section.

Common mistake

Pressing Enter repeatedly to get to a new page. When a line is added to the previous page, all the empty lines shift down by one; the heading lands in the middle of the page or an empty page opens in front of it. A page break inserted with `Ctrl+Enter`, on the other hand, keeps the following content at the top of the new page however much the text changes.

3.1.6.3 Making a single page landscape with a section break

The most typical use of a section break is to make a single page landscape inside a long portrait document. A wide table of measurements does not fit on a portrait page. The Orientation command, however, applies to the section the cursor is in. If the document is a single section, all pages become landscape. The solution is to put the table in a section of its own.

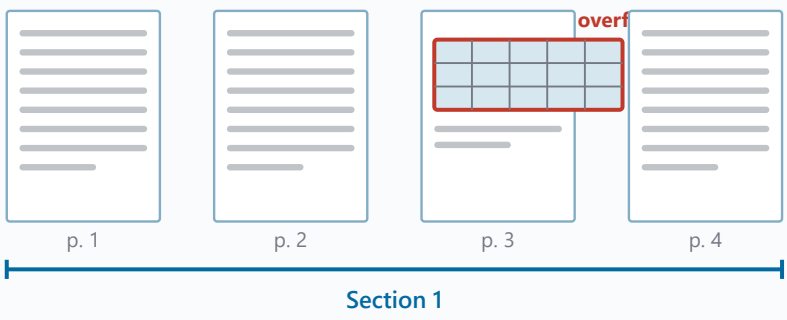


Figure 3.1.25: Starting point: the document is a single section (Section 1) and all four pages are portrait. The wide table on page 3 overflows the right edge.

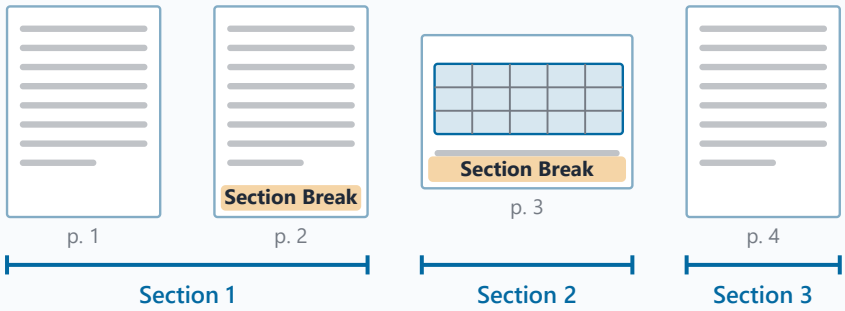


Figure 3.1.26: A Next Page section break was inserted before and after the table; with the cursor in Section 2, Orientation → Landscape was chosen. Only page 3 became landscape, and the table fits.

Example 3.1.15

In a four-page document the wide table is on page 3. The goal is to make only page 3 landscape. The steps and the section structure at each step are as follows (Figure 3.1.25 and Figure 3.1.26):

Step	Action	Sections	Landscape pages
0	start	S1: p.1–4	none
1	cursor before the table; Layout → Breaks → Next Page	S1: p.1–2 · S2: p.3–4	none
2	cursor after the table; Next Page again	S1: p.1–2 · S2: p.3 · S3: p.4	none
3	with the cursor in the table, Layout → Orientation → Landscape	same	only p.3

What happens if step 2 is skipped? The orientation in step 3 applies to the whole of S2, that is, to pages 3 and 4. If a page break had been used instead of a section break, the document would stay a single section and the orientation change would apply to all four pages. k section breaks divide a document into $k + 1$ sections; isolating a single page takes two section breaks.

Margins, number of columns, header, and page-number format are changed per section in the same way. If you want a command to affect only part of a document, the first question is: is that part a section of its own?

3.1.6.4 Headers, footers, and page numbers

Definition 3.1.18

The **header** and **footer** are areas repeated in the top and bottom margins of every page. Information such as the document title, section name, author, date, and page number is written there. They are edited with the Insert → Header / Footer commands or by double-clicking the top or bottom margin of the page.



Figure 3.1.27: A three-page report. The cover has no header or number (different first page); the following pages have the header “Report · Section 1” and an automatic page number (2, 3) in the footer.

The page number is inserted as a **field** from the Insert → Page Number menu (Figure 3.1.27). A field is a value that Word calculates and shows according to the current situation. A number field inserted once in the footer shows that page's number on every page. As pages are added and deleted, the numbers update by themselves. A “3” typed by hand, on the other hand, stays “3” on every page.

There are two more options for headers and footers. When **Different First Page** is checked, the first page of the section gets a separate header/footer area; this area is left empty so that the cover shows no number. **Different Odd & Even Pages** lets you put the number on the right on odd pages and on the left on even pages in documents printed double-sided.

3.1.6.5 Different First Page and Link to Previous

Once a document is divided into several sections, a new rule comes into play for headers. A new section starts with its header and footer *linked* to those of the previous section by default.

Definition 3.1.19

Link to Previous is the button on the Header & Footer contextual tab that appears while a header or footer is being edited. When it is on, the section's header shares the same content as the previous section's: a change made in one appears in the other. When it is turned off, the section has its own independent header.

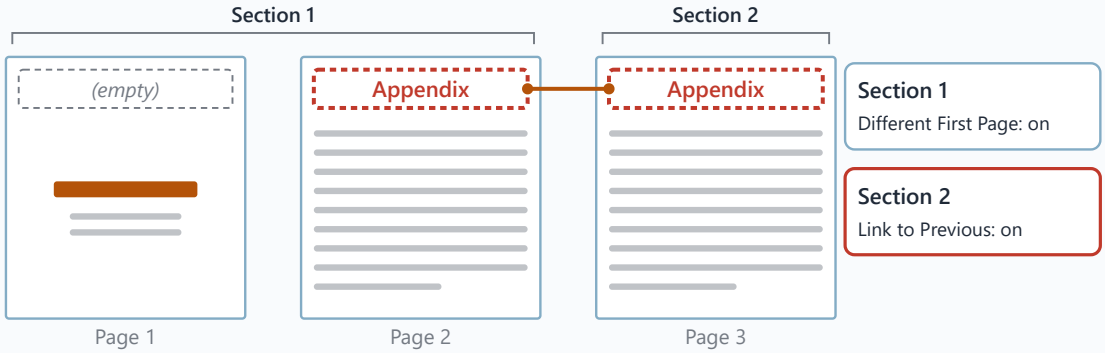


Figure 3.1.28: The wrong way. With Link to Previous on, the header of Section 2 was changed to "Appendix". Because the header is shared by the two sections, page 2 became "Appendix" too. The cover (Different First Page) stayed empty.

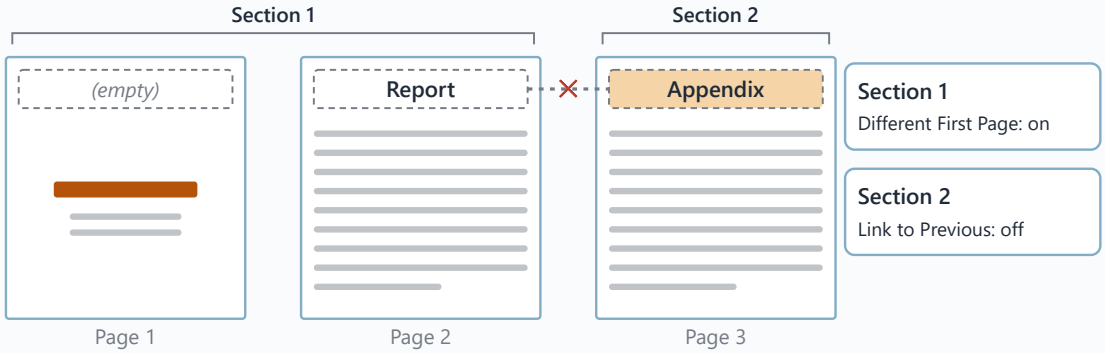


Figure 3.1.29: The right way. Link to Previous was first turned off in Section 2 (✕), and then "Appendix" was typed. Only page 3 says "Appendix"; page 1 is empty and page 2 says "Report".

Example 3.1.16

The header of a three-page report is "Report". No header is wanted on the cover; page 3 is an appendix and its header must read "Appendix". The steps and the header of the three pages at each step are as follows (Figure 3.1.28 and Figure 3.1.29):

Step	Action	p.1	p.2	p.3
1	single section; "Report" is typed in the header	Report	Report	Report
2	Different First Page is checked, the cover field is left empty	(empty)	Report	Report
3	a Next Page section break at the start of page 3; in S2 Link to Previous is on	(empty)	Report	Report
4a	wrong: with the link on, "Appendix" is typed in S2	(empty)	**Appendix**	Appendix
4b	right: 4a is undone; in S2 Link to Previous is turned off, then "Appendix" is typed	(empty)	Report	Appendix

The result of step 4a looks surprising: although only page 3 was edited, page 2 changed too. The reason is that linked headers are a single piece of content. The cover was not affected, because with Different First Page it has its own separate area. The rule is short: break the link *before* typing different content.

Common mistake

Changing the header of a new section right after adding a section break. Because Link to Previous is on, the change spreads backward to all linked sections. Before editing the new section's header, check the state of the Link to Previous button on the contextual tab.

3.1.6.6 Page numbering by section

In theses and long reports, page numbering has two parts. Front matter such as the cover, abstract, and table of contents is numbered with lowercase Roman numerals (i, ii, iii), and the main text is numbered 1, 2, 3 starting from the Introduction page. This arrangement is built with a section break and section-specific page-number settings.

Definition 3.1.20

The **Page Number Format** dialog is opened with Insert → Page Number → Format Page Numbers and manages two section settings. The **number format** is how the number is written: 1, 2, 3 or i, ii, iii. **Page numbering** is where the number starts: “Continue from previous section”, or the value typed in the “Start at” box. Both settings apply to the section the cursor is in.

A reminder about lowercase Roman numerals is useful: i = 1, v = 5, x = 10. If a smaller symbol comes before a larger one it is subtracted, and if it comes after it is added. So ii = 2, iii = 3, iv = 5 − 1 = 4, vi = 5 + 1 = 6.

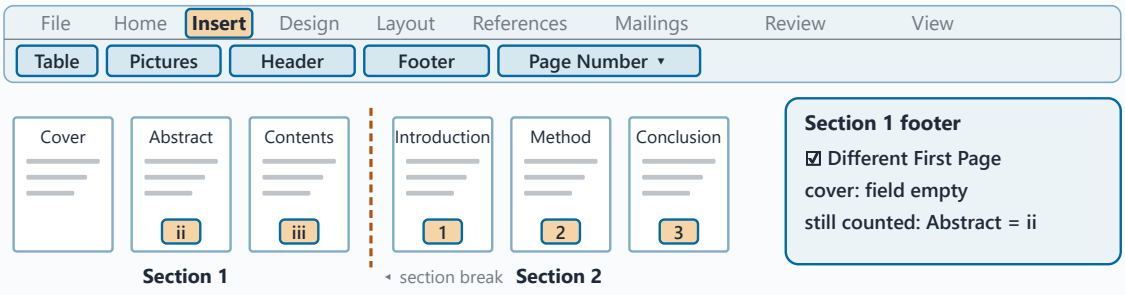


Figure 3.1.30: A report with two sections. Section 1 (Cover, Abstract, Contents) uses lowercase Roman numerals with Different First Page on: no number appears on the cover, but it is counted; Abstract is ii, Contents is iii. Section 2 (Introduction, Method, Conclusion) starts from 1.

Example 3.1.17

In a six-page report the first three pages are the Cover, Abstract, and Contents; the last three are the Introduction, Method, and Conclusion. The steps, and the number shown in the footer of the six pages after each step, are as follows (Figure 3.1.30):

Step	Action	Cover	Abstract	Contents	Introduction	Method	Conclusion
1	a Next Page section break at the end of the contents	–	–	–	–	–	–
2	Page Number → Bottom of Page	1	2	3	4	5	6
3	cursor in S1; number format i, ii, iii	i	ii	iii	4	5	6
4	cursor in S2; Start at: 1	i	ii	iii	1	2	3
5	Different First Page in S1	(none)	ii	iii	1	2	3

In step 3, Section 2 still counts from 4, because the format belongs to the section and only Section 1 was changed. Section 2 is set to “Continue from previous section”. Step 4 breaks that link. In step 5 no number appears on the cover but it is counted: Abstract is still ii, not i.

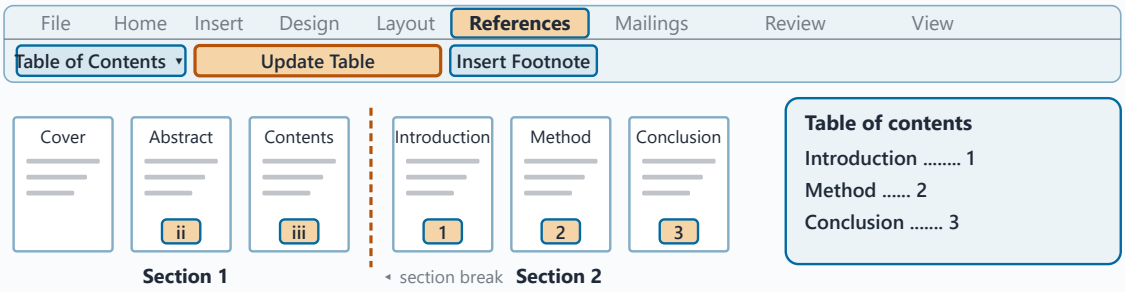


Figure 3.1.31: After the numbers changed, the table of contents still showed the old values (4, 5, 6). With References → Update Table it became Introduction 1, Method 2, Conclusion 3.

One last step is needed (Figure 3.1.31). If a table of contents was inserted earlier in the document, it does not update by itself when the numbering changes; it keeps showing the old numbers. The table of contents is also a field and is refreshed with the Update Table command (3.1.9.1).

3.1.7 Tables and pictures

In engineering reports, numerical results are given in tables, and setups and graphs as pictures. Word places these objects inside the text and offers separate contextual tabs for each. This section explains the structure of tables and how pictures relate to the text.

3.1.7.1 Tables

Definition 3.1.21

A **table** is an object that arranges information in cells made of rows and columns; it is inserted with Insert → Table. A **row** is a horizontal sequence of cells and a **column** a vertical one; the box where the two intersect is called a **cell**. The first row, which explains what the columns mean, is called the **header row**.

	Exam scores		
Student	Midterm	Final	Average
Ali	70	80	75
Ece	90	85	87.5
Can	60	72	66

merged cell
header row
row
cell
column

Figure 3.1.32: The elements of a table. At the top is a merged cell ("Exam scores"), below it the header row (Student, Midterm, Final, Average), and three data rows: Ali 70, 80, 75; Ece 90, 85, 87.5; Can 60, 72, 66.

A few principles help when building a table's layout (Figure 3.1.32):

- A **header row** should always be present. If the table runs over several pages, you can choose to repeat the header row on every page (table Layout tab → Repeat Header Rows).
- **Merging cells** should be used only for a meaningful group of headings. Too many merged cells make it hard for screen readers to read the table correctly and break sorting.
- **Table styles** (Table Design tab) apply borders and colors consistently to the whole table; there is no need to shade cells one by one.
- **Alignment**: text is aligned left, and numbers right or on the decimal point, so that the digits line up.

Example 3.1.18

The Average column in Figure 3.1.32 is the arithmetic mean of the midterm and final scores. Let us check the values:

1. Ali: (70 + 80) ÷ 2 = 150 ÷ 2 = 75.
2. Ece: (90 + 85) ÷ 2 = 175 ÷ 2 = 87.5.
3. Can: (60 + 72) ÷ 2 = 132 ÷ 2 = 66.

All three values are correct. Word tables are not spreadsheets: when a score changes, the average does not update by itself. The Formula command on the table Layout tab can do simple calculations, but calculation-heavy tables are prepared in Excel and inserted into Word.

3.1.7.2 Pictures

Pictures are inserted with Insert → Pictures from the computer, from online sources, or from stock images. When a picture is selected, the **Picture Format** contextual tab opens. Three tools on this tab are especially important:

- **Alt text**: a short text that explains the purpose of an image to screen-reader users who cannot see it. It should describe the information the image conveys, not be a file name like “image1.png” (3.1.11.1).
- **Crop**: leaves unneeded edge areas of the image outside the frame. The cropped part is not deleted but hidden; it can be brought back if necessary. To reduce file size, the cropped areas can also be deleted.
- **Layout Options**: the small button that appears at the top-right corner when a picture is selected. It changes the picture's relationship to the text, that is, the text-wrapping type.

3.1.7.3 Text wrapping types

Definition 3.1.22

Text wrapping is the setting that determines how a picture is laid out with the text around it. **In line** (called “In Line with Text” in Word) places the picture on the line like a character of the text. With the other types the picture becomes an object independent of the text, one that can be moved freely on the page.

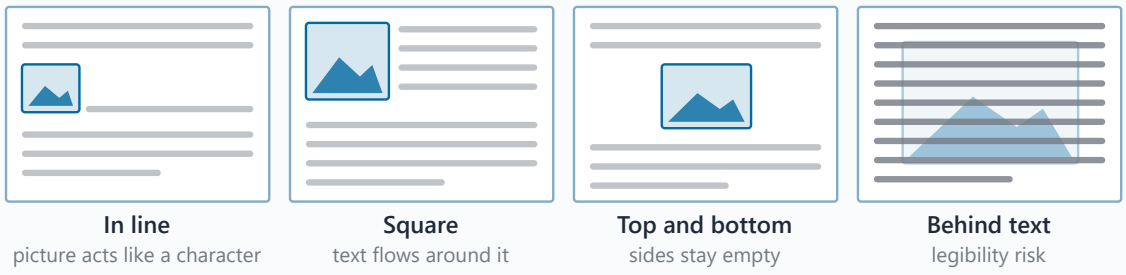


Figure 3.1.33: The four text-wrapping types. In line: the picture acts like a character. Square: the text flows around the picture. Top and Bottom: the sides of the picture stay empty. Behind Text: the picture sits under the text; legibility risk.

Type	Behavior	Suitable use
In Line with Text	sits on the line like a character and enlarges the line height; its position depends on the text	figures that stand in their own paragraph in reports; the most stable type
Square	text flows around the rectangular frame of the picture	brochures, newsletters
Tight / Through	text flows close to the picture's actual outline	pictures with a transparent background and irregular shapes
Top and Bottom	text stays only above and below the picture; the sides are empty	pictures close to the page width
Behind Text	the picture is on a layer beneath the text; the text is read over it	watermark, faded background logo
In Front of Text	the picture is on a layer above the text; it covers the text beneath	rarely; it can hide text

Table 3.1.11: Text-wrapping types and where they are used.



Figure 3.1.34: The same picture with four wrapping types (row by row, starting at top left): in line (line height grows), square (text flows around it), behind text (text is read over it), in front of text (the text beneath is invisible).

Figure 3.1.34 compares the four types on the same picture. In reports the safest choice is in line. The picture sits in its own paragraph, can be centered, and can have a caption added below it. As text is added and deleted, the picture flows along with the text. With free types such as Square or Tight, the

picture is anchored to the page. When the text in front of it grows longer, the picture can stay where it is and end up on a different page from its paragraph.

Example 3.1.19

A faded logo will go in the background of a club announcement; the text must be readable over the logo. Which type is chosen, and how is the logo prepared?

1. **Type.** The logo must be on the layer beneath the text: **Behind Text**. In Front of Text covers the text, Square pushes the text around the logo, and in line puts the logo on a line.
2. **Fading.** From the Picture Format → Color menu, a light tone or "Washout" is chosen. If the logo is not faded, the legibility of the text drops.
3. **Alternative route.** If the logo will repeat in the same place on every page, a picture watermark is added with Design → Watermark → Custom Watermark. In that case Word does the fading and positioning itself.

3.1.8 Review and collaboration

After a document is written it is reviewed at least once: spelling errors are corrected, wording is tidied, and someone else's suggestions are weighed. The Review tab gathers the tools for this work. Some of the tools are for the author's own checking, others for several people working on the same document.

3.1.8.1 Language, proofing, and word count

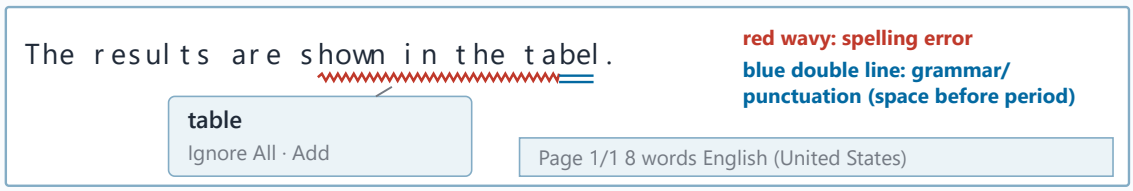


Figure 3.1.35: Spell checking. In the sentence "The results are shown in the tabel .", "tabel" is marked with a red wavy line (spelling error), and the space before the period with a blue double line (grammar or punctuation). Right-clicking offers the suggestion "table". The status bar shows "8 words" and "English (United States)".

While you type, Word checks the text in the background (Figure 3.1.35). A word not found in the dictionary is marked with a red wavy line, and a suspected grammar or punctuation problem with a blue double line. Right-clicking a marked word lists suggestions. A suggestion can be chosen, the mark can be ignored, or the word can be added to the user's dictionary.

The checking rules are determined by the **document language**. If the status bar says “English (United States)”, the English dictionary is used. If English text is accidentally marked as Turkish, nearly every word is underlined in red. The language is changed by selecting the text and using Review → Language → Set Proofing Language. This way, English paragraphs and a Turkish abstract can sit side by side in one document, each checked with its own rules.

Tool	Role
Editor	collects spelling, grammar, and style suggestions in a single pane
Word Count	shows the number of words, characters (with and without spaces), paragraphs, lines, and pages
Thesaurus	offers synonyms for the selected word
AutoCorrect	fixes common typing errors as you type (predefined matches such as “teh” → “the”)

Table 3.1.12: Review tools.

Word count is a criterion often required in assignments. The status bar shows the word count of the whole document; if some text is selected, it also gives the count for the selection, as in “120 of 2,350 words”. The Review → Word Count window gives a detailed count and asks whether footnotes should be included in the count.

Common mistake

Treating text with no underlines as error-free. The check catches only words that are not in the dictionary. Typing “100 kΩ” instead of “10 kΩ” in “The resistor value in the circuit diagram is 10 kΩ”, writing “there” for “their”, or leaving a sentence unfinished often goes unflagged. Automatic suggestions do not know the context either; the author judges each suggestion before accepting it.

3.1.8.2 Comments and Track Changes

Imagine a student sending a thesis draft to the advisor. If the advisor edits the text directly, the student cannot see what changed. If the advisor writes notes in the margin by hand, the notes get mixed with the text. Word offers two separate tools for this collaboration.

Definition 3.1.23

A **comment** is a question, suggestion, or piece of feedback attached to a spot without changing the text; it is added with Review → New Comment and appears as a balloon in the right margin. With **Track Changes** on, every insertion, deletion, and format change is recorded along with who made it and when. A recorded change stays pending until the author **accepts** or **rejects** it.

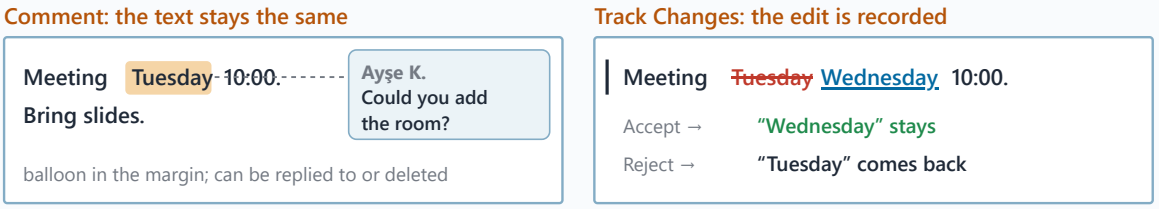


Figure 3.1.36: Top, a comment: the text "Meeting Tuesday 10:00. Bring slides." stays the same, and Ayşe K.'s question "Could you add the room?" sits in a balloon in the margin. Bottom, Track Changes: the deleted "Tuesday" appears struck through and the inserted "Wednesday" underlined. With Accept, "Wednesday" stays; with Reject, "Tuesday" comes back.

Figure 3.1.36 shows the difference between the two tools. A comment is a conversation: it can be replied to, marked as "resolved", or deleted. It never touches the text. Track Changes, by contrast, is a suggestion in the text itself: if accepted, the text becomes final; if rejected, it returns to its old state. Changes can be accepted one by one or all at once with Accept → Accept All Changes.

3.1.8.3 Markup views

A document with tracked changes can be shown on screen at different levels of detail. These views are chosen from the list in the Review → Tracking group. A view changes only the display; it does not touch the state of pending changes.

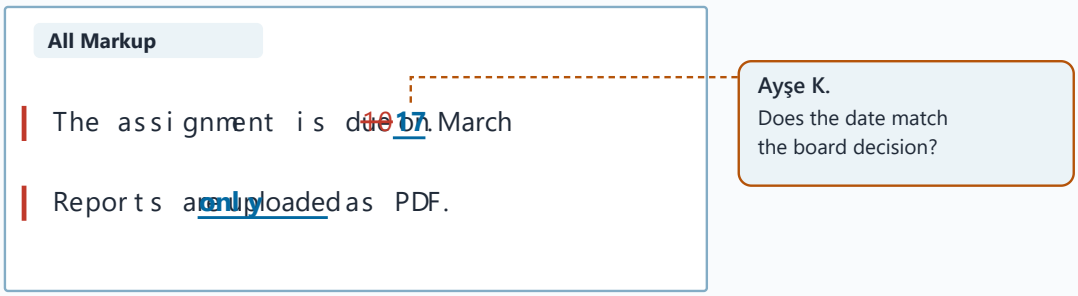


Figure 3.1.37: The All Markup view. "10" is deleted (struck through), "17" and "only " are inserted (underlined); there are change bars in the margin and Ayşe K.'s comment: "Does the date match the board decision?"

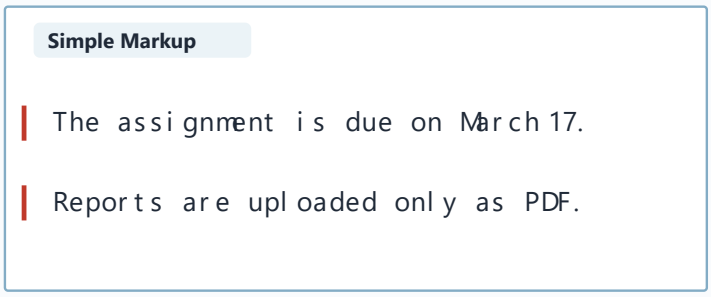


Figure 3.1.38: The same document in the Simple Markup view: the text appears in its final form, but the changes are still pending.

View	What it shows
Simple Markup	shows the text as if the changes had been applied; only a small bar is left in the margin
All Markup	shows deletions struck through, insertions underlined and colored, and comments in balloons
No Markup	shows the text in its final form, with no marks at all
Original	shows the text as it was before the changes

Table 3.1.13: Markup views.

Example 3.1.20

The document has two lines: “The assignment is due on March 10.” and “Reports are uploaded as PDF.” Track Changes is turned on and the following operations are performed:

Step	Action	Line 1 in All Markup	Pending
1	“10” is deleted, “17” is typed	The assignment is due on March ~~10~~ 17.	2
2	on line 2, “only ” is inserted before “as PDF”	same	3
3	the date is selected, New Comment	same; a balloon in the margin	3 + 1 comment
4	switch to Simple Markup	The assignment is due on March 17.	3 + 1 comment
5	Accept All Changes; the comment is deleted	The assignment is due on March 17.	0

In steps 4 and 5 the text on screen is the same, but the state of the document is different. In step 4 the changes are only hidden. If the document is sent in this state, the recipient can switch to All Markup and see the deleted “10”. In step 5 the changes have been made part of the document and the comment has been deleted.

Common mistake

Sending a document that looks clean in the Simple Markup or No Markup view as if it were the final version. Hiding is not accepting. Before sharing, check on the Review tab that no pending changes or comments remain; File → Info → Check for Issues → Inspect Document

also lists them.

3.1.8.4 Accepting and rejecting changes one by one

When the suggestions of several people have been gathered in the same document, each suggestion is evaluated separately. The **Next** and **Previous** buttons on the Review tab move between pending changes. **Accept and Move to Next** makes the current change final and moves on to the next; **Reject and Move to Next** undoes the change and moves on to the next. The next change is chosen by its position in the text, not by who made it or when.

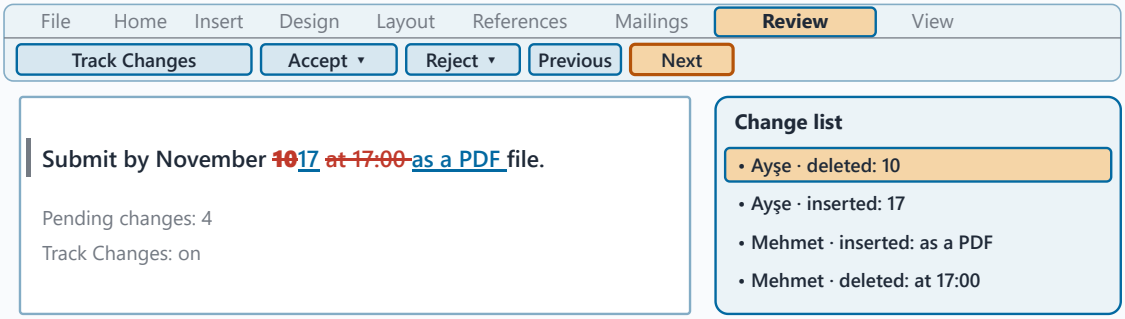


Figure 3.1.39: Starting point: four changes by Ayşe and Mehmet are pending. Change list: Ayşe deleted “10”, Ayşe inserted “17”, Mehmet inserted “as a PDF”, Mehmet deleted “at 17:00”.

Example 3.1.21

Text: “Submit by November ~~10~~17 ~~at 17:00~~ as a PDF file.” The time must stay in the text; the other three changes are correct. Starting with Next, the following operations are performed at each step (Figure 3.1.39):

Step	Selected change	Command	Conclusion	Pending
0	—	—	—	4
1	Ayşe: “10” deleted	Accept and Move to Next	“10” is gone from the text	3
2	Ayşe: “17” inserted	Accept and Move to Next	“17” became ordinary text	2
3	Mehmet: “at 17:00” deleted	Reject and Move to Next	deletion undone; the phrase stays in the text	1
4	Mehmet: “as a PDF” inserted	Accept	insertion made final	0
5	—	Track Changes is turned off	later edits are not recorded	0

Final text: “Submit by November 17 at 17:00 as a PDF file.” In step 3 the next change is not Mehmet's insertion but his deletion, which comes earlier in the text. *Rejecting* a deletion brings the deleted text back; rejecting an insertion removes the inserted text.

Forgetting to turn Track Changes off is also common. As long as the button stays on, the author's own corrections pile up as pending changes too. At the final read-through they too have to be accepted one by one.

3.1.9 Long documents

In a document of more than ten pages, readers need guides to find what they are looking for: a table of contents, numbered headings, numbered figures, footnotes, and a bibliography. All of these elements can be typed by hand. But as the document changes, every hand-typed number becomes a source of errors. Word generates all of these elements automatically from the structure of the document and keeps them up to date.

3.1.9.1 Heading styles and the table of contents

Building an automatic table of contents is a four-step flow: heading styles are applied, the levels are checked in the navigation pane, the table of contents is inserted, and the table is updated as the document changes.

Definition 3.1.24

A **table of contents** is a list inserted with References → Table of Contents and generated automatically from the heading styles in the document (Heading 1, Heading 2, Heading 3). Each line holds the heading's text and its page number. The table is a **field**: it does not refresh by itself when the document changes; it is updated with the **Update Table** command (or with `F9` while the table is selected).

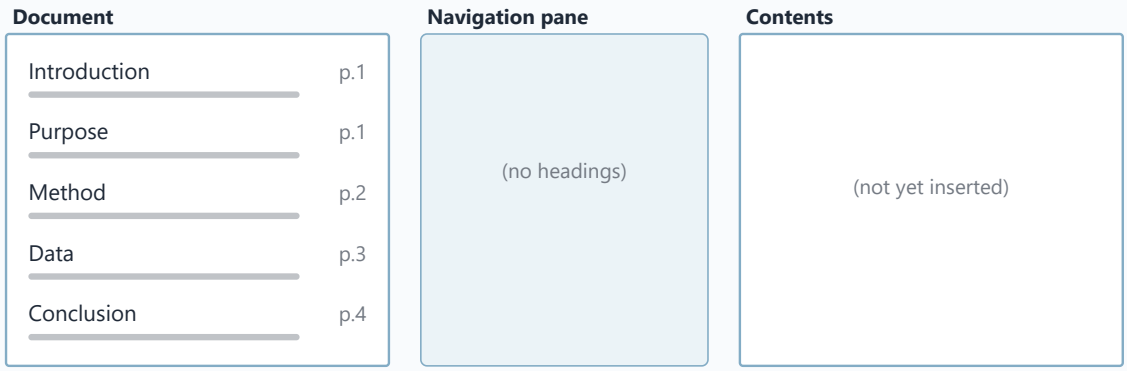


Figure 3.1.40: The headings are merely typed bold and large, in the Normal style. Word does not recognize them as headings; the navigation pane is empty.

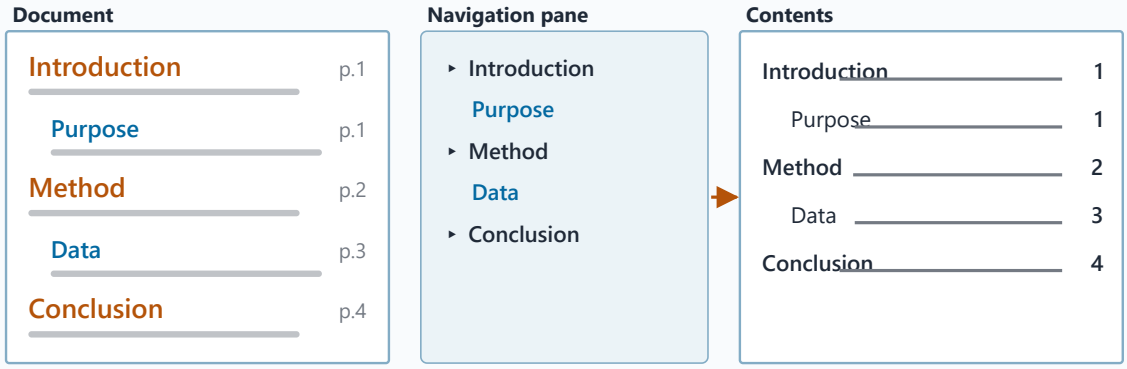


Figure 3.1.41: Heading 1 was applied to Introduction, Method, and Conclusion, and Heading 2 to Purpose and Data; the navigation pane shows the two-level structure. The table was generated automatically with References → Table of Contents: Introduction 1, Purpose 1, Method 2, Data 3, Conclusion 4.

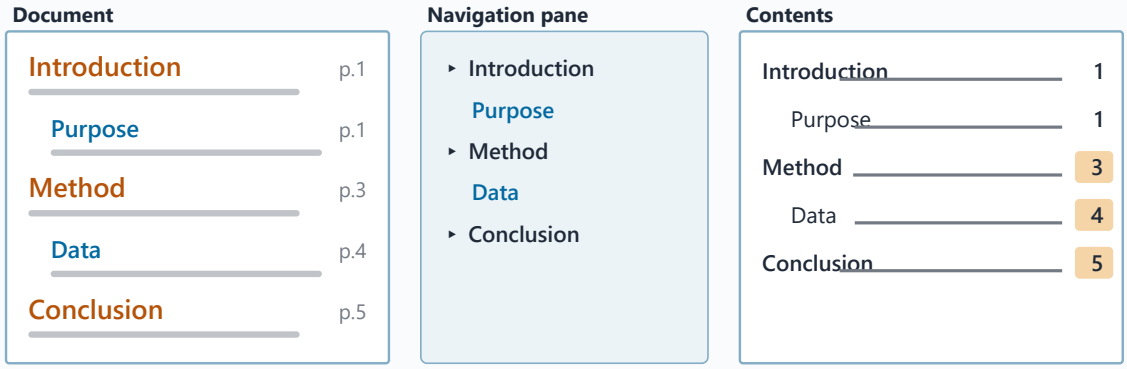


Figure 3.1.42: One page of text was inserted before Method and Update Table (F9) was applied: Method 3, Data 4, Conclusion 5.

Example 3.1.22

The report has five headings: Introduction and Purpose on page 1, Method on page 2, Data on page 3, Conclusion on page 4. The steps, and the navigation pane and table of contents at each step, are as follows (Figure 3.1.40, Figure 3.1.41, and Figure 3.1.42):

Step	Action	Navigation pane	Contents (Introduction, Purpose, Method, Data, Conclusion)
1	headings in Normal style, bold	empty	none
2	Heading 1 for Introduction, Method, Conclusion	Introduction · Method · Conclusion	none
3	Heading 2 for Purpose and Data	Introduction › Purpose · Method › Data · Conclusion	none
4	References → Table of Contents	same	1, 1, 2, 3, 4
5	a page of text is inserted before Method	same	1, 1, 2, 3, 4 (old)
6	Update Table	same	1, 1, 3, 4, 5

The inconsistency in step 5 is deliberate design. Fields are not recalculated at every keystroke; updating is left to the user. The navigation pane, by contrast, shows headings instantly. Update Table offers two options. **Update page numbers only** corrects the numbers of the existing lines. **Update entire table** also reflects newly added, deleted, or renamed headings in the table. If a new heading has been added, the second option is needed.

Common mistake

Typing the table of contents by hand, or forgetting to update the automatic table after inserting it. In both cases the numbers are wrong when the document is submitted. Before exporting to PDF or printing, all fields are updated: select the document with `Ctrl+A` and press `F9`.

3.1.9.2 Multilevel heading numbering and thesis format

In theses and technical reports the headings are numbered: 1 Introduction, 1.1 Purpose, 1.1.1 Objectives. If these numbers are typed into the heading text by hand, then when a section is inserted in the middle, all the later numbers must be corrected by hand. The right way is to link a multilevel list to the heading styles.

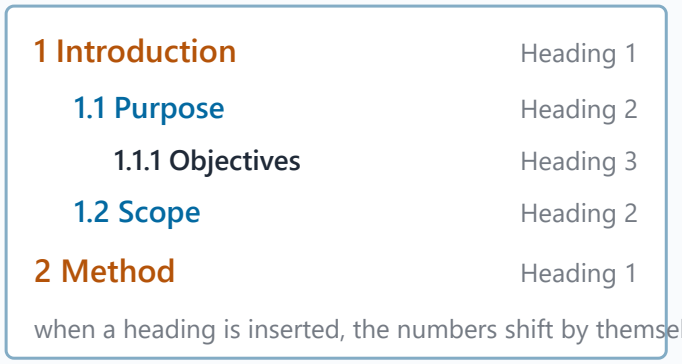


Figure 3.1.43: Multilevel numbering linked to the heading styles: 1 Introduction (Heading 1), 1.1 Purpose (Heading 2), 1.1.1 Objectives (Heading 3), 1.2 Scope (Heading 2), 2 Method (Heading 1). When a heading is inserted, the numbers shift by themselves.

The linking is done in the Home → Paragraph → Multilevel List → Define New Multilevel List window. In the window, “More >>” opens additional options and, for each level, a style is chosen in the “Link level to style” field: level 1 to Heading 1, level 2 to Heading 2, level 3 to Heading 3. After that, applying Heading 2 to a paragraph gives it both the heading look and the correct number (Figure 3.1.43).

Example 3.1.23

In the structure of Figure 3.1.43, a heading “Limitations” in the Heading 2 style is added not between “1.1 Purpose” and “1.1.1 Objectives” but after “1.1.1 Objectives”. How do the numbers change?

1. The new heading is the second Heading 2 of section 1: **1.2 Limitations**.
2. The old “1.2 Scope” becomes the third Heading 2 of the same section: **1.3 Scope**.
3. “2 Method” is a Heading 1 and does not change. When the table of contents is updated, the new numbers are reflected there too.

Had the numbers been typed by hand, “1.2 Scope” would have to be changed to “1.3” by hand, and references in the text such as “as described in 1.2” would have to be hunted down one by one.

The formatting rules for theses vary by institution. Commonly seen values are given in Table 3.1.14. These are only examples; margins, font, and citation format differ from institution to institution. The valid rules are in the institution's writing guide.

Topic	Common example	Where is it set?
Margins	about 2.5 cm on each side; the gutter may be separate	Layout → Margins
Font and size	Times New Roman or similar, 12 pt	definition of the Normal style

Topic	Common example	Where is it set?
Line spacing	1.5 or double; may differ in headings	Paragraph dialog (in the style definition)

Table 3.1.14: Formatting values commonly seen in theses and reports, and where they are set in Word (example values).

The “Where is it set?” column of the table reflects a principle: values such as font and spacing are given not to individual paragraphs but in *the definitions of styles*. If the institution's rule changes, only the style definition is corrected.

3.1.9.3 Footnotes, citations, and the bibliography

Definition 3.1.25

A **footnote** is an additional explanation of an expression, given at the bottom of the page. An **endnote** gathers the explanations at the end of the document or section. Both are inserted from the References tab and numbered automatically. A **citation** is a reference in the text to a source. A **bibliography** is the list of the sources cited in the document.

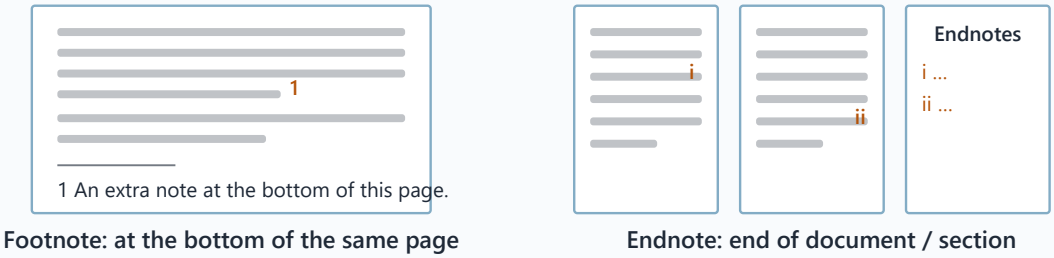


Figure 3.1.44: Footnote and endnote. The footnote sits at the bottom of the same page, below the separator line (1). Endnotes are gathered at the end of the document or section and by default are numbered with lowercase Roman numerals (i, ii).

To insert a footnote, place the cursor at the end of the word the explanation attaches to and choose References → Insert Footnote. Word puts a superscript number in the text and moves the cursor to the footnote area at the bottom of the page (Figure 3.1.44). When a new footnote is inserted in between, the numbers of the following footnotes shift by themselves. If, as the text moves, the word a footnote is attached to goes onto another page, the footnote moves with it to the bottom of that page.

Word can also take over source management. With References → Insert Citation → Add New Source, a source's author, title, year, and publisher are entered in a form. This source is inserted as a citation wherever wanted in the text. The **citation style** (APA, IEEE, MLA, Chicago, and so on) determines the format of these citations and of the bibliography lines. When the style is changed, all citations and the bibliography switch to the new format. The References → Bibliography command generates an automatic list from the cited sources.

Example 3.1.24

A lab report cites two sources: a textbook and a datasheet. The instructor first asks for APA, then says to switch to IEEE format.

1. Each source is entered once with the Add New Source form. At the citation points, the relevant source is chosen from the Insert Citation list.
2. With the citation style set to APA, the citations in the text take the author–year form, for example "(Patterson, 2020)".
3. At the end of the report a Bibliography is inserted; the two sources are ordered by APA rules.
4. IEEE is chosen from the References → Style list. The citations in the text turn into numbers in square brackets, for example "[1]". When the bibliography is updated, its lines also switch to the numbered IEEE format.

With hand-typed citations this switch means retyping every citation and every bibliography line one by one. Even so, the lines Word generates are only as correct as the information entered in the form. The author of the document checks the accuracy of author names, years, and page numbers.

3.1.9.4 Captions and cross-references

Definition 3.1.26

A **caption** is a label that identifies a figure, table, or equation with a numbered title such as "Figure 1: Circuit diagram"; it is inserted with References → Insert Caption. A **cross-reference** is a dynamic link from the text to a heading, figure, table, or page elsewhere in the document. A **table of figures** is a list of figures generated from the captions.

The number in a caption is a field too. When a new figure is added to the document, Word shifts the numbers of the following figures. If the phrase "as seen in Figure 3" in the text was typed by hand, it knows nothing of this shift and starts pointing to the wrong figure. A cross-reference (References → Cross-reference), by contrast, takes the number from the caption and shows the new number when updated.

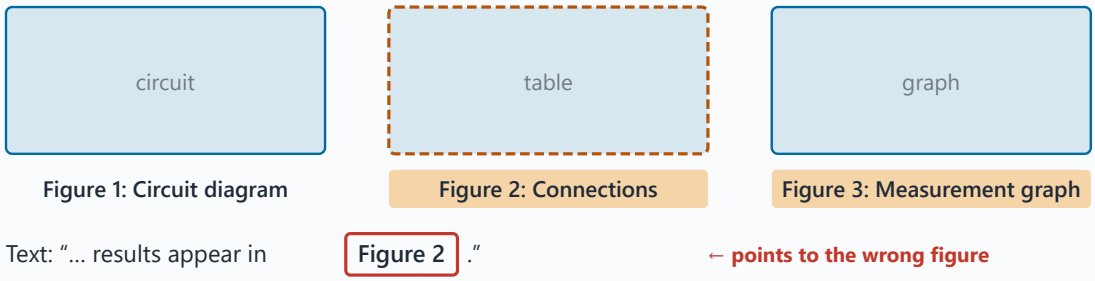


Figure 3.1.45: A new table image and caption (Figure 2: Connections) were inserted between the two figures. The graph's number became Figure 3, but because the cross-reference in the text has not been updated yet, it still says "Figure 2" and points to the wrong figure.

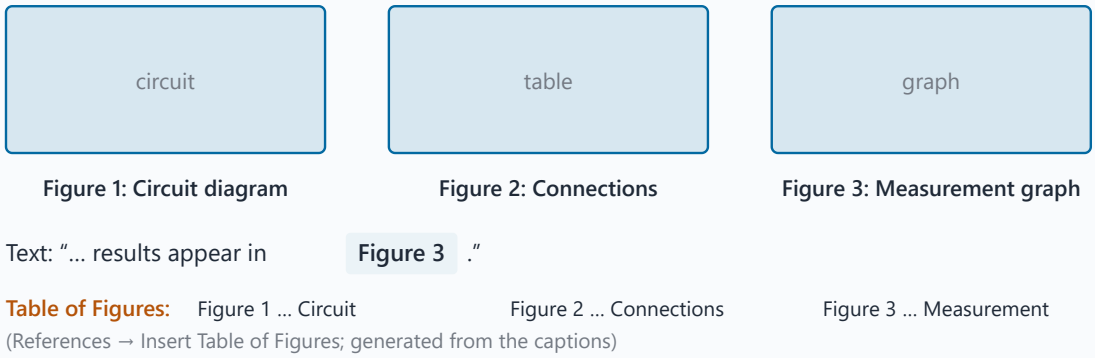


Figure 3.1.46: After `Ctrl+A` and `F9` the cross-reference reads "Figure 3". A table of figures was generated from the captions with References → Insert Table of Figures.

Example 3.1.25

The document has two figures: "Figure 1: Circuit diagram" and "Figure 2: Measurement graph". In the text "the results appear in Figure 2", "Figure 2" is a cross-reference to the graph. A new figure is inserted between the two (Figure 3.1.45 and Figure 3.1.46):

Step	Action	Captions	Reference in the text
1	start	1 Circuit · 2 Measurement	Figure 2 (correct)
2	a "Connections" figure and caption are inserted between them	1 Circuit · 2 Connections · 3 Measurement	Figure 2 (points to the wrong figure)
3	`Ctrl+A`, `F9` (update fields)	same	Figure 3 (number correct)
4	the wording around the reference is checked	same	Figure 3 (no change needed)
5	References → Insert Table of Figures	same	same; three rows in the table of figures

In step 3 the cross-reference updates only the number. The words around it are ordinary text typed by hand, not part of the field, and Word does not rewrite them; a phrase such as “the second figure” would stay wrong. So after the fields are updated, the text around each cross-reference is read through once (step 4).

Common mistake

Typing numbers such as “Figure 3” by hand in a caption or in the text. When a figure is added or deleted, all the numbers and references have to be corrected one by one; if one is missed, the reader looks at the wrong figure. Because captions and cross-references are fields, a single `F9` updates them all.

3.1.10 Mail merge

The department office will send each of 120 students a letter announcing their final grades. The text is the same for everyone; only the name, department, and grade change. Writing 120 letters one by one would take hours and risks putting the wrong grade in the wrong student's letter. Mail merge solves this with a main document and a list.

3.1.10.1 The main document, the data source, and fields

Definition 3.1.27

Mail merge is a Mailings-tab feature that combines a **main document** with a **data source** to produce a personalized output for each recipient. The main document holds the text and layout common to everyone. The data source (for example, an Excel table) holds the recipients' information: each **column** is a **field**, and each **row** is a **record**, that is, one recipient. A **merge field** is a placeholder such as «Name» that marks where a value will go in the main document.

The merge proceeds in five stages: the main document is prepared, the recipient list is chosen, merge fields are inserted, the results are previewed, and finally the documents or e-mails are produced. The preview stage should not be skipped. A typo in a column name or an empty cell means an error in hundreds of documents at once.

The first row of the data source holds the *field names*: Name, Department, Grade. This row is not a recipient and produces no letter. So a list of n rows whose first row is a header yields $n - 1$ letters. Short, space-free, meaningful field names make it easier to pick the right field in the main document.

3.1.10.2 Merging step by step

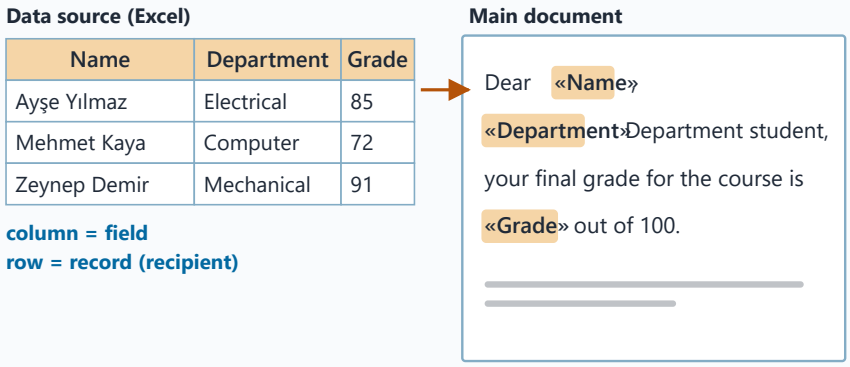


Figure 3.1.47: On the left, the data source (Excel): the Name, Department, and Grade columns and three records (Ayşe Yılmaz, Electrical, 85; Mehmet Kaya, Computer, 72; Zeynep Demir, Mechanical, 91). On the right, the main document: the «Name», «Department», and «Grade» fields were placed in the blanks.

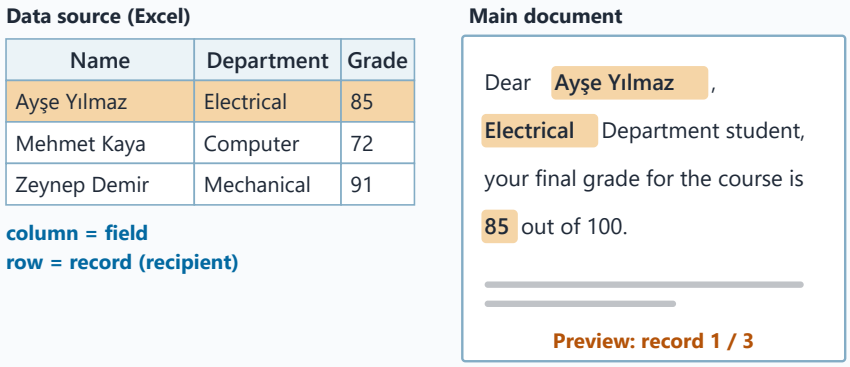


Figure 3.1.48: Preview Results, record 1 / 3: the values of record 1 have replaced the fields: "Dear Ayşe Yılmaz, Electrical Department student, your final grade for the course is 85 out of 100."

Example 3.1.26

Main document: "Dear _____, _____ Department student, your final grade for the course is _____ out of 100." The data source is the three-record Excel table in Figure 3.1.47. The steps and the look of the main document at each step are as follows:

Step	Command on the Mailings tab	What the main document shows
1	(the main document is typed)	Dear _____, _____ Department … _____ …
2	Select Recipients → Use an Existing List (Excel file)	same; 3 recipients connected
3	Insert Merge Field: Name, Department, Grade	Dear «Name», «Department» Department … «Grade» …

Step	Command on the Mailings tab	What the main document shows
4	Preview Results, record 1	Dear Ayşe Yılmaz, Electrical Department … 85 …
5	next record (2)	Dear Mehmet Kaya, Computer Department … 72 …
6	next record (3)	Dear Zeynep Demir, Mechanical Department … 91 …
7	Finish & Merge → Edit Individual Documents	a new document with 3 letters is created

The Excel file has 4 rows: 1 header row and 3 records. The output is 4 − 1 = 3 letters. During the preview the common text of the main document never changes; only the values that replace the fields change. The Finish & Merge menu also lets the same merge go directly to the printer or be sent as e-mail. The Envelopes and Labels commands on the Mailings tab produce address labels and envelopes from the same data source.

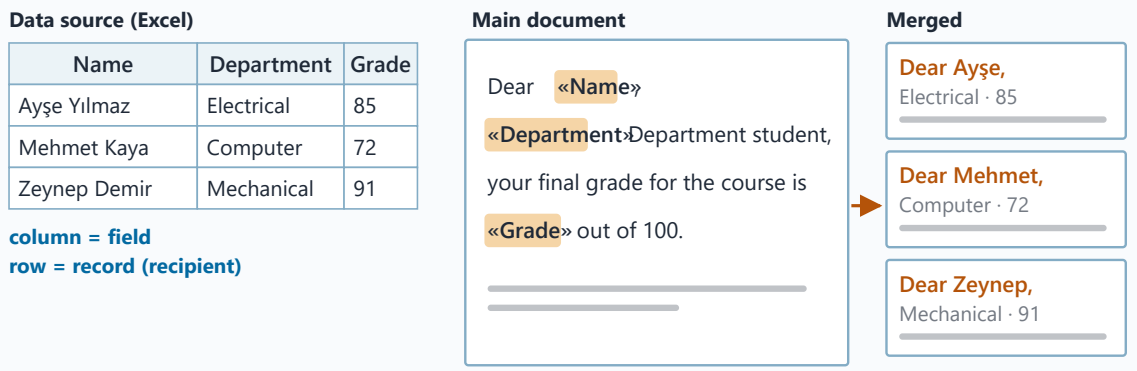


Figure 3.1.49: Finish & Merge: one letter was produced for each record, 3 documents in all (Ayşe – Electrical · 85, Mehmet – Computer · 72, Zeynep – Mechanical · 91).

Common mistake

Starting a merge with an Excel list that has no header row. Word counts the first row as field names; the first student's name “Ayşe Yılmaz” becomes a field name and no letter is produced for that student. The first row of the data source is always the field names. Another common mistake is numbers not arriving in their Excel format. For example, a grade such as 87.5 may appear in the letter with different digits. The preview exposes such problems in the first records.

3.1.11 Quality and sharing

After a document has been written and reviewed, it goes through a final check. Is the document accessible to everyone who will read it? Does it keep its layout when printed or exported to PDF? Does it contain a forgotten comment, hidden change, or personal information? This section deals with these questions and with the security risk that macros carry.

3.1.11.1 Accessible documents

Definition 3.1.28

An **accessible document** is one that people with visual, hearing, or motor impairments can also read and navigate using assistive technologies (for example, **screen readers** that read text aloud). The **accessibility checker** (Review → Check Accessibility) lists the possible problems in the document and suggests fixes.

A screen reader reads a document aloud from top to bottom. The user can jump from heading to heading, list the links, and move through a table cell by cell. All of these capabilities rest on the structure of the document. A document that looks right but lacks structure is a meaningless heap of text to a screen reader. The principles of accessibility therefore coincide with the good habits taught since the beginning of these notes:

- **Heading styles** tell the screen reader the structure of the document; the user can jump between headings.
- **Alt text** explains what meaningful images convey. Images that are purely decorative are marked as “decorative” and are not read out.
- **Link text** describes the destination. “Click here” says nothing once it is taken out of context; “Registration Guide (PDF)” names the destination.
- **Table structure** is kept simple and the header row is clear. Layouts made with empty cells or merged cells confuse reading across cells.
- **Color and contrast**: meaning is not conveyed by color alone. Instead of “Rows in red failed”, a word or symbol saying “failed” is added to the rows as well. Enough contrast is left between text and background.



Figure 3.1.50: Unclear versus clear usage. On the left, the link “For details, click here” and the alt text “image1.png”. On the right, the link “Details: Registration Guide (PDF)” and the alt text “Enrollments rise from 2020 to 2024”.

Figure 3.1.50 shows two small differences. Good alt text conveys to someone who cannot see the image *the information the image gives*. For a chart, writing “chart” is not enough. The trend the chart shows (“Enrollments rise from 2020 to 2024”) is what gets written. If the detailed numbers are already given in the text or in a table, the alt text can be kept short.

Common mistake

Treating accessibility as an extra job that concerns only readers with disabilities. Heading styles, meaningful link text, and simple tables make every reader's work easier. The bookmarks that appear in a document exported to PDF, search engines indexing the content correctly, and the document being readable on a phone all rest on the same structure.

3.1.11.2 Printing and PDF

The File → Print screen shows the print settings on the left and the **print preview** on the right. The preview is the last check of how the document will come out on paper. An overflowing table, an empty page that has slipped in, or a picture stuck to the margin is noticed here.



Figure 3.1.51: The File → Print screen. On the left Copies: 1, Printer: Microsoft Print to PDF, Print All Pages, Pages: 1-3, 5, Print on Both Sides, Portrait Orientation · A4; on the right, the print preview of a 6-page document.

The settings in Figure 3.1.51 mean the following. **Copies** is how many times each page is printed. The **Pages** box determines which pages are printed: a hyphen marks a range, a comma separates individual pages. For example, “1-3, 5” is pages 1, 2, 3, and 5; that is, 4 pages are printed. The Print All

Pages menu also lets you print the selected text or only the current page. **Print on Both Sides** uses both sides of the paper on printers that support it and cuts paper consumption roughly in half.

Example 3.1.27

In the 6-page document of Figure 3.1.51, “1-3, 5” is typed in the Pages box. How many pages come out, and how many sheets of paper are used, for one-sided and two-sided printing?

1. **Pages to be printed.** “1-3” is three pages (1, 2, 3), and “5” is one page. In total 4 pages; pages 4 and 6 are not printed.
2. **One-sided.** Each page is printed on a separate sheet: 4 sheets.
3. **Two-sided.** Each sheet holds two pages: 4 ÷ 2 = 2 sheets. If the number of pages were odd, it would be rounded up; for example, 5 pages would take 3 sheets.
4. **If Copies were 3.** Two-sided printing would use 3 × 2 = 6 sheets.

The standard way to share a document without changes is PDF. **Exporting** (File → Export → Create PDF/XPS Document, or Save As → file type PDF) produces a copy that keeps the layout. This copy looks the same on a device that does not have Word. In the options window you can ask for PDF bookmarks to be created from the headings. A clickable table of contents then appears in the sidebar of the PDF reader. This is another gain from a document written with heading styles.

Before sharing, the document is also checked for hidden content. Comments, pending changes, and **document properties** (metadata such as author name, title, and tags) go with the document. The File → Info → Check for Issues → **Inspect Document** command lists them and, if wanted, removes them all at once. An assignment whose author field shows someone else's name gives away that an old file was used instead of a template.

Common mistake

Printing or exporting to PDF without looking at the print preview. A paragraph added at the last minute can push a table onto the next page. An empty page may be left at the end of the document, or the table of contents may show old numbers. A preview takes a few seconds; reprinting costs paper and time.

3.1.11.3 Macros and security

Definition 3.1.29

A **macro** is a small program that makes a repeated sequence of commands runnable with a single command. It is created by recording the user's actions (View → Macros → Record Macro) or by writing code in **VBA** (Visual Basic for Applications). Documents that contain macros are stored with the `.docm` extension, and templates that contain macros with `.dotm`.

Macros save time in real work. For example, applying the same format to hundreds of tables, or setting all the pictures in a document to the same width, can be done with a single macro. But a macro is a program. A macro that runs when a document is opened can delete files, download other files, or install malware on the computer. That is why attackers hide malicious macros in “invoice” or “shipping information” documents attached to e-mails.

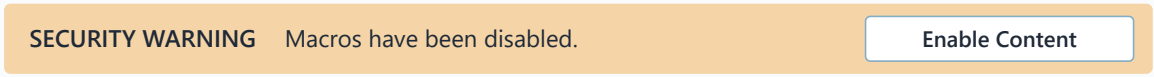


Figure 3.1.52: The security warning bar shown when a document with macros is opened: “SECURITY WARNING Macros have been disabled.” and the Enable Content button.

When opening a document with macros, Word by default does not run the macros and shows the warning in Figure 3.1.52. Pressing the Enable Content button means giving the macros in the document permission to run on the computer. For files downloaded from the internet or received by e-mail, current Microsoft 365 versions can block macros entirely. In that case a security-risk warning appears in place of the button. The rule is simple: do not enable content in a document of unknown or unexpected origin. For an unexpected macro-enabled document, even one that seems to come from a known person, ask the sender through a separate channel.

3.1.11.4 A document-preparation workflow and common mistakes

A long document is prepared in six stages: the purpose and audience are determined, a draft structure (the headings) is built, the content is written, styles and images are arranged, the document is reviewed, and it is published. The order has an important logic: structure and content come first, and detailed formatting is left for last. Fussing over fonts while the content is still changing is wasted effort. If styles are used, the formatting is in any case set in one place, at the very end, in a few minutes.

The common mistakes spread through these notes are gathered in Table 3.1.15. The fix for all of them rests on the same principle: tell Word the structure, not the appearance.

Mistake	Consequence	Right way
pressing Enter repeatedly for a new page	empty lines shift when text is added, and the page layout breaks	`Ctrl+Enter` or Layout → Breaks
aligning with the space bar	the alignment shifts when the font changes	tab, indent, or table
typing a heading merely bold and large	the navigation pane and table of contents do not see the heading	Heading 1 / Heading 2 style
typing numbers by hand (1., 1.1, Figure 3)	all the numbers must be corrected by hand after an insertion	list, multilevel list, caption
typing the table of contents and page numbers by hand	wrong numbers remain when pages shift	automatic table of contents, page-number field
pressing Enter at the end of a line	every line becomes a separate paragraph and justification breaks	Word wraps the line itself; Enter only at the end of a paragraph

Table 3.1.15: Common Word mistakes, their consequences, and the right ways.

If the structure is built right, the appearance can be changed in one place: the look of headings comes from a style definition, the page layout from section settings, and the numbers from fields. Even if the document grows to a hundred pages, a change requested at the last minute takes a few minutes.

3.1.12 Summary

- Word is a word processor; it stores a document in a `.docx` file. Commands sit on the ribbon in a tab → group → command arrangement; contextual tabs appear when a table or picture is selected.
- Save updates the open file; Save As creates a new file and the window then works on that file. AutoSave works only on a file in the cloud; AutoRecover is no substitute for saving.
- When a template (`.dotx`) is opened, an untitled copy derived from it is opened; the template does not change.
- The structural levels of a document are character, word, paragraph, section, and document. Character formatting applies to the selection, paragraph formatting to the paragraph the cursor is in, and page settings to the section the cursor is in.
- Cut moves, Copy duplicates; the clipboard keeps its content after pasting. Replace All also changes the insides of words; “Find whole words only” or a Find Next preview is needed.
- Hidden marks (, ·, →) are shown with the button; in Find and Replace they are searched with the codes `^p` , `^t` , `^l` , `^s` , `^m` . `^p^p` → `^p` is repeated until the result is 0.
- A style is a named package of formatting; when its definition changes, all the items that use it are updated. Direct formatting takes precedence over the style; `Ctrl+Space` removes manually applied character formatting.
- A page break does not create a section; a section break does. To make a single page landscape, put a section break before and after it.

- The header of a new section starts linked to the previous one by Link to Previous; break the link before typing different content. The page-number format and starting value are set per section.
- Heading styles are the source of the table of contents, captions are the source of the table of figures, and cross-references depend on both. These are fields; they are refreshed with `F9` or Update Table.
- A comment does not change the text; Track Changes records every edit as a suggestion awaiting acceptance or rejection. Hiding markup is not accepting the changes.
- In mail merge the main document holds the common text and the data source holds the recipients; the header row does not produce a letter.
- Before sharing, check hidden content with print preview, the accessibility check, and Inspect Document; do not enable macros in a document of unknown origin.

3.1.13 Key terms

document ribbon tab contextual tab navigation pane view save / save as AutoSave AutoRecover version history template cursor selection clipboard Format Painter Find and Replace formatting marks non-breaking space bookmark hyperlink font point superscript / subscript alignment justify indent hanging indent line spacing paragraph spacing multilevel list style theme direct formatting margin orientation page break section break header / footer table, cell, header row text wrapping alt text watermark comment track changes markup table of contents footnote / endnote citation, bibliography caption cross-reference table of figures mail merge merge field accessibility checker print preview macro

---

3.1.14 Test Yourself

1. `design.docx` is opened on disk. In order: paragraph A is typed, `Ctrl+S`; paragraph B is typed, Save As → `design_final.docx`; paragraph C is typed and Word is closed without saving. In the end, which paragraphs are in the two files?

   Answer

   `design.docx`: only A, because nothing was written to this file after Save As. `design_final.docx`: A and B. C was lost because it was never saved; after Save As, the file open in the window was `design_final.docx`.

2. A one-line document: “This note is read before the exam.” The cursor is at the start of the line. The following are pressed in order: `Ctrl+→`, `Ctrl+Shift+→`, `Ctrl+Shift+→`, `Ctrl+I`. Which words

become italic, and where does the cursor end up?

Answer

`Ctrl+→` takes the cursor to the start of “note”. Two `Ctrl+Shift+→` presses select first “note ”, then “note is ”. `Ctrl+I` applies to the selection: “note is” becomes italic. The cursor is at the start of the word “read” and the selection remains.

3. After the “Conclusion” paragraph there are four empty paragraphs in a row, followed by the “Appendices” paragraph. With Find what `^p^p` and Replace with `^p`, how many times is Replace All run, and how many replacements are made each time?

Answer

After “Conclusion.” five follow in a row. Run 1: pairs are matched from the left, 2 replacements; 3 remain (two empty paragraphs). Run 2: 1 replacement; 2 remain (one empty paragraph). Run 3: 1 replacement; a single remains. Run 4 gives 0 results and shows the job is done.

4. The headings Purpose, Data, and Scope are in the Heading 2 style (12 pt, blue). Data is made orange from the ribbon. Then the definition of Heading 2 is changed with Modify… to 13 pt and purple. How do the three headings look? What happens next if Data is selected and `Ctrl+Space` is pressed?

Answer

Purpose and Scope: 13 pt, purple. Data: 13 pt but orange, because the color was given to it by hand and direct formatting takes precedence over the style; the size comes from the definition because it was not given by hand. `Ctrl+Space` deletes the manually applied color; Data also becomes 13 pt and purple.

5. In a 10-page portrait document, a wide table spanning pages 4 and 5 and a diagram on page 9 must be landscape. How many section breaks are needed, and how many sections does the document consist of?

Answer

One section break before page 4, after page 5, before page 9, and after page 9: 4 section breaks. k section breaks make $k + 1$ sections: p.1–3, p.4–5, p.6–8, p.9, and p.10, that is, 5 sections. The sections made landscape are sections 2 and 4.

6. In a thesis the first four pages are the Cover, Approval, Abstract, and Contents; these are Section 1 and use lowercase Roman numerals, with Different First Page on. Section 2 starts at 1 on the Introduction page. Which number appears on the Abstract page and on the 7th physical page of

the document?

Answer

In Section 1 the cover is counted but its number is not shown: Cover (i, not shown), Approval ii, Abstract iii, Contents iv. “iii” appears on the Abstract. Section 2 starts at 1 on the 5th physical page; the 7th physical page is number 3.

7. With Track Changes on, in the sentence “The lab report is due on Friday.”, “Friday” is deleted and “Monday” typed, and “by noon ” is inserted before “on”. The author accepts the deletion and the insertion of “Monday” and rejects the insertion of “by noon ”. What is the final sentence? If the author had sent the document in Simple Markup without accepting anything, what could the recipient have seen?

Answer

Final sentence: “The lab report is due on Monday.” The rejected insertion is removed from the text. Had it been sent without accepting anything, three changes would have been pending; the recipient could have switched to All Markup and seen the deleted “Friday” and the inserted phrases. Hiding is not accepting.

8. The Excel file used in a mail merge has 31 filled rows; the first row is the field names. The Grade cell of two students is empty. How many letters does Finish & Merge produce, and what happens to the empty cells?

Answer

The header row produces no letter: 31 − 1 = 30 records, 30 letters. Records with empty cells also produce letters; the place of the «Grade» field is left blank. That is why the records are previewed before the results are produced and the missing data is filled in.

9. On the Print screen of a 10-page document “2-4, 7, 9-10” is typed in the Pages box, 2 in the Copies box, and Print on Both Sides is chosen. How many sheets of paper are used?

Answer

The pages to be printed are 2, 3, 4, 7, 9, and 10: 6 pages. Each sheet holds 2 pages when printed two-sided: one copy takes 6 ÷ 2 = 3 sheets. Two copies take 2 × 3 = 6 sheets.

10. On an A4 page the left margin (with gutter) is 3.5 cm and the other margins are 2.5 cm. How many millimeters wide is the text area? With line spacing “Exactly 24 pt”, what is the maximum number

of lines that fit on a page?

Answer

The width is 210 − 35 − 25 = 150 mm, the height 297 − 25 − 25 = 247 mm. 24 pt = 24 × 25.4 ÷ 72 ≈ 8.47 mm. 247 ÷ 8.47 ≈ 29.2; rounded down: 29 lines.

3.1.15 Further reading

- Microsoft Support, *Word help and learning* (support.microsoft.com/en-us/word): short articles and videos for every feature of Word, written with the English interface names; the most reliable source for verifying the current location of the commands mentioned in these notes.
- Microsoft Support, *Make your Word documents accessible to people with disabilities*: explains alt text, heading structure, tables, and color use step by step, together with the Accessibility Checker.
- M. Butterick, *Practical Typography* (practicaltypography.com): a free online book that explains, with reasons, the choices of font, line spacing, margins, and alignment that make a document readable.

3.1.16 References

1. Microsoft Support, *Basic tasks in Word*.
2. Microsoft Support, *Track changes in Word*.
3. Microsoft Support, *Templates, themes and Word styles*.
4. Microsoft Support, *Make your Word documents accessible to people with disabilities*.
5. M. Butterick, *Practical Typography*, online book, practicaltypography.com.

PART I · CHAPTER 3.2

Microsoft Excel

The measurements of a lab experiment, a course's grade list, a project's materials budget, or a shop's stock record can all be organized the same way: records go in rows and attributes in columns. Spreadsheet programs combine this layout with calculation. A formula typed into a cell produces a result from the values in other cells and updates that result automatically when those values change. These notes start from the basic concepts of Microsoft Excel, such as the workbook, the cell, and the range. They then cover formulas, cell references, commonly used functions, data-management and analysis tools, and charts. The notes use the tab and command names of the English interface of Microsoft 365 and the English function names. Arguments are separated by commas, as in English-language Excel. The result of every formula is shown calculated step by step.

LEARNING OUTCOMES

Workbook

Use the concepts of workbook, worksheet, cell, row, column, and range correctly; enter data of the appropriate type into a cell and explain that a number format changes only the display.

Formulas

Calculate formula results according to operator precedence; predict in advance how relative, absolute, and mixed references change when copied; compute with basic, logical, conditional, and lookup functions; find and fix the causes of error values.

Data management

Apply tables, sorting, filtering, SUBTOTAL, conditional formatting, PivotTables, and data validation.

Visualization

Choose the chart type that suits the purpose of the data and format the chart so that it does not mislead; prepare a workbook for printing, sharing, and protection.

3.2.1 Excel and the worksheet

Excel works on a grid of rows and columns. Each box of the grid can hold a number, a piece of text, or a formula. This simple structure suits very different jobs, from a small shopping list to thousands of rows of measurement data. This section explains the parts of the grid and what they are called.

3.2.1.1 What is a spreadsheet for?

Consider a grade sheet kept on paper. Each student's midterm and final grades are written down, and then the averages are worked out one by one on a calculator. If a grade was entered wrongly, that student's average is recalculated. The class average is found again from scratch. A spreadsheet turns this around. The average is written once as a formula; when one of the grades changes, the result refreshes by itself.

Definition 3.2.1

A **spreadsheet** is a program that holds data in cells arranged in rows and columns and calculates from that data with formulas written in the cells. Microsoft Excel, LibreOffice Calc, and Google Sheets are common examples of the kind.

The main uses of Excel can be grouped under six headings:

- **Calculation.** Calculations done with formulas and functions are repeatable. If the same calculation is to be done again with new data, only the data is changed.
- **List management.** Records of students, products, customers, or measurements are kept in order in rows and columns.
- **Analysis.** Records are sorted, those that meet a condition are filtered out, and totals and averages are produced by group.
- **Visualization.** Numbers are turned into charts; trends and comparisons are seen at a glance.
- **Planning.** Budgets, calendars, stock, and task tracking are run in table form.
- **Sharing.** A workbook stored in the cloud can be edited by several people at the same time, who can also add comments to cells.

For an engineering student the most important of these is often calculation. The current of a circuit at different resistor values, the mean of an experiment's measurements, or the total cost of a bill of materials can be found in Excel with a few formulas. Once a formula is written correctly, it can be copied down hundreds of rows.

3.2.1.2 Workbook, sheet, and cell

Every file opened in Excel is a **workbook**. Inside the workbook are one or more **worksheets**. Each sheet is a separate table with the same grid structure. The sheet names appear on tabs at the bottom of the window; when a new workbook is opened there is a single sheet, named “Sheet1”.

Definition 3.2.2

A **workbook** is the whole Excel file; today its default extension is `.xlsx`. A **worksheet** is a single grid of cells inside the workbook. A **cell** is the single box where a row and a column intersect, and it holds a single value or formula.

In the Excel window there are several important elements around the grid. Figure 3.2.1 shows them together.

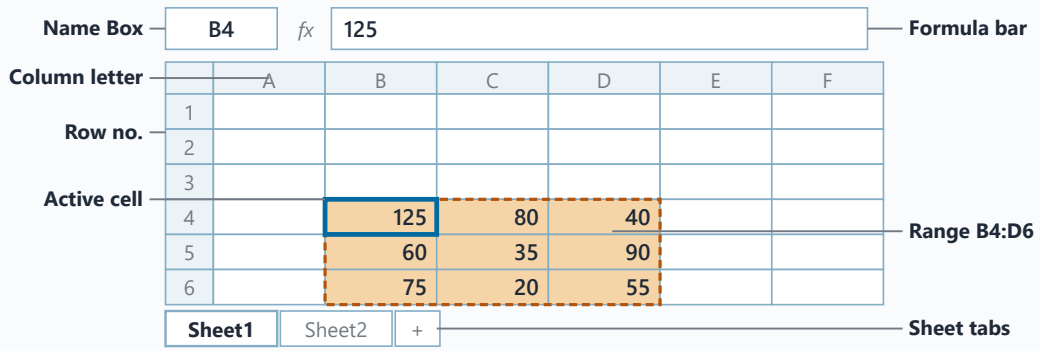


Figure 3.2.1: The basic elements of the Excel window. The Name Box at the top left shows the address of the active cell (B4), and the formula bar next to it shows that cell's content (125). Columns are named with letters and rows with numbers. The range B4:D6 covers nine cells; the sheet tabs are at the bottom.

- **Name Box**: shows the address of the selected cell. Typing an address here and pressing Enter goes to that cell. For example, typing `Z500` is faster than scrolling with the mouse for a long time.
- **Formula bar**: shows the actual content of the active cell. If the cell holds a formula, the cell shows the result and the formula bar shows the formula itself.
- **Column letters and row numbers**: the headers along the edges of the grid. Clicking a header selects the whole column or row.
- **Active cell**: the cell that is currently selected and into which what you type will go. It is shown with a bold border.
- **Sheet tabs**: list the sheets in the workbook. The **+** button next to the tabs adds a new sheet.

At the top of the window is the **ribbon**. The ribbon divides commands into tabs: **File**, **Home**, **Insert**, **Page Layout**, **Formulas**, **Data**, **Review**, and **View**. In these notes the location of a command is written as “Data → Sort”. This notation means that the Data tab is chosen first, and then the Sort

button on that tab.

3.2.1.3 Cell address and range

Every cell has an address. The address is written with the column letter first, then the row number: C3 is the cell where column C and row 3 intersect. Column letters run from A to Z; after Z two-letter names begin: AA, AB, …, AZ, BA. In Microsoft 365 Excel a sheet has 1,048,576 rows and 16,384 columns. The last column is named XFD.

Example 3.2.1

Which columns from the start are AB and XFD? How many cells are there on a sheet in all?

1. **A single letter.** A = 1, B = 2, …, Z = 26. Column letters resemble a number system with 26 digits.
2. **AB.** The first letter counts "rounds" of 26, and the second letter gives the position within the round: A × 26 + B = 1 × 26 + 2 = **28**. So AA is column 27 and AB is column 28.
3. **XFD.** X = 24, F = 6, D = 4. With three letters the first letter counts rounds of 26 × 26 = 676: 24 × 676 + 6 × 26 + 4 = 16,224 + 156 + 4 = **16,384**. This number is 2^{14}.
4. **Number of cells.** 1,048,576 rows (2^{20}) × 16,384 columns = **17,179,869,184** cells, that is, about 17 billion cells.

These limits are rarely stretched in practice. Even so, data with millions of rows does not fit on one sheet. For data of that size, other tools such as a database or a programming language are used.

Most calculations are done not with a single cell but with a group of cells. A rectangular region made of adjacent cells is called a **range**.

Definition 3.2.3

A **range** is a rectangular block of cells determined by its top-left and bottom-right corner cells, and is written with a colon between the two addresses: `B4:D6` covers the nine cells from B4 to D6.

Example 3.2.2

How many cells do the ranges `B4:D6`, `A1:A10`, and `C3:F10` contain?

1. **Formula.** Number of columns = last column − first column + 1; number of rows = last row − first row + 1. The number of cells is the product of the two. The "+ 1" comes from both end cells being included in the range.

2. **B4:D6.** Columns B, C, D: 4 − 2 + 1 = 3. Rows 4, 5, 6: 6 − 4 + 1 = 3. Number of cells 3 × 3 = **9**.
3. **A1:A10.** A single column, 10 − 1 + 1 = 10 rows: **10** cells.
4. **C3:F10.** C, D, E, F: 4 columns. 10 − 3 + 1 = 8 rows. 4 × 8 = **32** cells.

There are several ways to select a range. The mouse can be dragged from the top-left cell to the bottom-right cell. Or the first cell is selected and, holding Shift, the last cell is clicked. Typing `B4:D6` in the Name Box gives the same result. Cells that are not adjacent are selected one by one while holding the Ctrl key.

3.2.1.4 A value in a cell, a formula in a cell

A cell can hold two kinds of content. The first is a value typed directly, such as the number 12 or the word “Pen”. The second is a formula that begins with an equals sign. A formula uses the addresses of other cells and calculates its result from the values in those cells. Figure 3.2.2 shows a small shopping table.

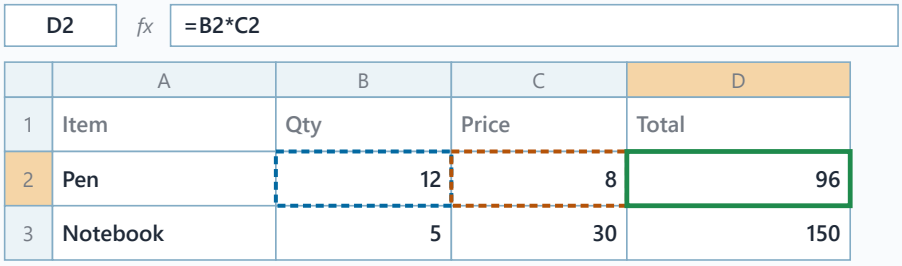


Figure 3.2.2: A table holding item, quantity, and price. D2 is selected: the cell shows the result (96) and the formula bar shows the formula itself (`=B2*C2`).

The formula `=B2*C2` is typed in cell D2. Excel reads this formula as: “multiply the value in B2 by the value in C2.” Since B2 holds 12 and C2 holds 8, D2 shows 12 × 8 = 96. By the same logic, the formula `=B3*C3` in D3 gives 5 × 30 = 150. The formula uses not the numbers themselves but the addresses of the cells where the numbers are. The real power of a formula comes from this distinction.

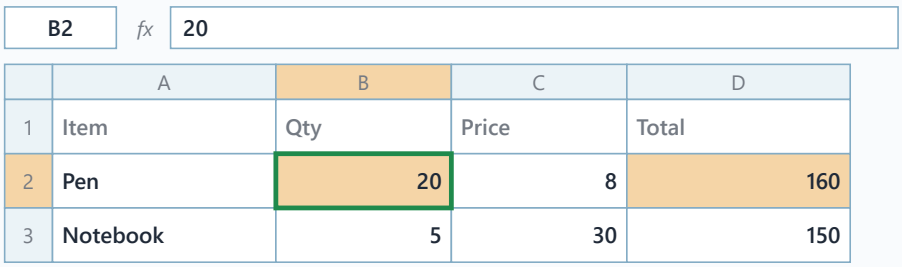


Figure 3.2.3: The quantity in B2 was raised from 12 to 20. Although the formula in D2 was not touched, the result became 160 by itself.

In Figure 3.2.3 the number of pens has been raised from 12 to 20. The formula in D2 is still `=B2*C2`, but because B2 changed, the result becomes 20 × 8 = 160. Had `=12*8` been typed in the formula instead of the cell references, the formula would have had to be corrected by hand when the quantity changed. For this reason, cell addresses are used in formulas instead of constant numbers wherever possible.

Common mistake

Typing the calculated result by hand instead of a formula. For example, if the number 96 is typed directly into D2, then when B2 changes, D2 stays at the old value. The cell looks the same, but the table is now wrong. Whether a cell holds a value or a formula can be told by looking at the formula bar.

3.2.2 Data entry

Excel interprets what is typed into a cell: “12” is taken as a number, “Pen” as text, and “2024-03-01” as a date. This interpretation determines whether calculations will come out right. This section covers data types, practical ways to enter data into a cell, and operations on the rows and columns of the sheet.

3.2.2.1 Data types

Definition 3.2.4

A **data type** is the kind of value in a cell, and it determines which operations can be done with that value. The main types in Excel are text, number, date and time, logical value, formula, and error value.

- **Text**: content that is not counted in calculations, such as a name, a description, or a category. “Pen”, “Electrical”, and “A-101” are each text.
- **Number**: a quantity that can be counted in calculations. In English settings the decimal separator is the period: 35.5 is a number. If 35,5 is typed, English-language Excel does not recognize it as a number and stores it as text; an entry such as 3/5, meanwhile, can even turn into a date (March 5).
- **Date and time**: Excel stores dates behind the scenes as numbers. That is how the difference in days between two dates is found by subtraction.
- **Logical value**: can be only `TRUE` or `FALSE`. The results of comparison formulas are of this type: the formula `=5>3` gives TRUE.
- **Formula**: a calculation that begins with an equals sign. The cell shows not the formula itself but its result.

- **Error value**: a special result that appears when a formula cannot be calculated. `#DIV/0!`, `#NAME?`, and `#VALUE!` are examples of this type.

Excel signals the type of what is entered in a cell by its alignment. If no special alignment has been given to the cell, text is left-aligned, numbers and dates are right-aligned, and logical values and error values are centered. Figure 3.2.4 shows these default alignments.

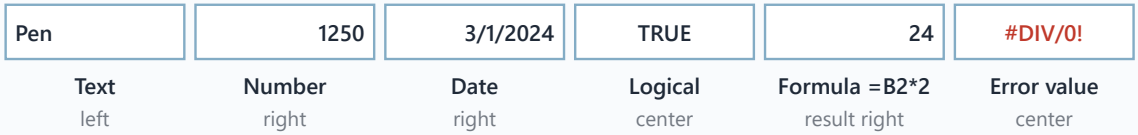


Figure 3.2.4: The default alignment of data types in a cell. Text is left-aligned, and numbers and dates right-aligned; the result of a formula is also right-aligned if it is a number. Logical values and error values are centered.

This rule is a small but valuable checking tool. If one of the numbers in a column sits left-aligned, that cell has most likely been stored as text. Such a cell is not included in the calculation of functions like SUM. The cause of a wrong total is often a cell like this.

Example 3.2.3

In an Excel with English (US) settings, `250`, `12.5`, `2024-03-01`, `12 TL`, and `TRUE` are typed in order into cells A1:A5. What are the type and alignment of each cell? What does `=A1+A2` give?

1. **A1 = 250.** It consists only of digits: a number, right-aligned.
2. **A2 = 12.5.** Because the decimal separator is the period, it is recognized as a number and right-aligned.
3. **A3 = 2024-03-01.** The year-month-day pattern is recognized as a date; since a date is also a number, it is right-aligned.
4. **A4 = 12 TL.** It has letters at the end; Excel cannot convert it to a number and stores it as text. It is left-aligned.
5. **A5 = TRUE.** It is recognized as a logical value and centered.
6. **Formula.** `=A1+A2` = 250 + 12.5 = **262.5**.

If you try to calculate with the “12 TL” in A4, a `#VALUE!` error appears. The right way to show a currency is to type only 12 in the cell and add the unit with a number format.

3.2.2.2 Entering and editing data

To enter data into a cell, select the cell, type, and press Enter. Enter confirms the entry and moves to the cell below; Tab moves to the cell on the right. To correct the content of a cell without deleting it, enter **Edit** mode. This mode is entered with the F2 key or by double-clicking the cell. In Edit mode the cursor is inside the cell, and the arrow keys move the cursor within the text instead of changing cells. Table 3.2.1 summarizes the keys used most often in data entry.

Action	Key	Description
Edit a cell	`F2`	Switches to Edit mode; the content is corrected without being deleted.
Confirm the entry	`Enter`	Enters the value and moves to the cell below.
New line inside a cell	`Alt+Enter`	Starts a second line in the same cell.
Cancel the entry	`Esc`	Cancels what was typed; the old content stays.
Clear the content	`Delete`	Deletes the value; the cell's format stays.
Select a row	`Shift+Space`	Selects the whole row of the active cell.
Select a column	`Ctrl+Space`	Selects the whole column of the active cell.

Table 3.2.1: Keys frequently used in data entry.

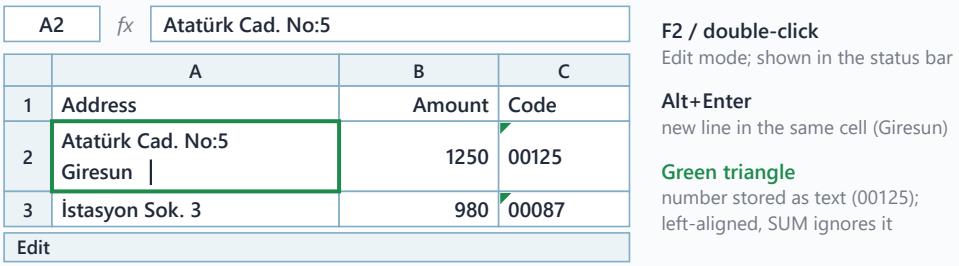


Figure 3.2.5: Cell A2 is in Edit mode; the address is split over two lines with Alt+Enter. The codes 00125 and 00087 in column C are stored as text. Excel marks these cells with a green triangle in the top-left corner.

Column C in Figure 3.2.5 shows an important case. Values such as a student number, a postal code, or a product code consist of digits, but the leading zeros are meaningful. If Excel takes 00125 as a number, it drops the leading zeros and shows 125. To keep the zeros, the value is stored as text: either an apostrophe is typed into the cell first (`'00125`) or the cell is formatted as **Text** beforehand. In this case Excel puts a green triangle in the top-left corner of the cell and gives the warning “Number stored as text”.

Storing values as text is right for codes that will not enter calculations. Numbers that will enter calculations must never be stored as text. Such cells also land in the wrong place when sorting, because text is sorted separately from numbers.

Common mistake

Using numbers copied from another program without checking them. Numbers that come from web pages or text files are often pasted as text; sometimes there is an invisible space at the end as well. The total of such a column comes out too low, and no error message appears. Left-aligned numbers and green triangles are the signs of this problem. It is fixed by choosing **Convert to Number** from the warning button next to the green triangle.

Two principles are worth following when entering data. The first is that a cell holds a single observation: instead of "12 kg", only 12 is typed and the unit moves to the column header. The second is that a column holds a single attribute: dates and descriptions are not mixed in the same column. These two principles are the precondition for sorting, filtering, and formulas to work without trouble.

3.2.2.3 Rows, columns, and sheets

As a table grows, operations on rows and columns become necessary. These are done both from the right-click menu and from the **Cells** group on the Home tab (Insert, Delete, Format).

- **Inserting and deleting.** When a row is inserted, the rows below it shift down; when a column is deleted, the columns to its right shift left. The addresses in formulas update by themselves to follow this shift.
- **Size.** Row height and column width are adjusted by dragging the line between the headers. Double-clicking the line widens the column automatically to fit its longest content.
- **Hiding.** A hidden row or column is not deleted; it is only removed from view. Formulas keep using hidden cells.
- **Freezing.** When you scroll down a long list, the header row leaves the screen and you forget which column is which. The **View → Freeze Panes → Freeze Top Row** command keeps row 1 fixed on the screen.
- **Sheet name and layout.** Instead of names like "Sheet1", short names that describe the content are used: "Data", "Calc", "Report". Raw data, calculation, and report can be kept on separate sheets.

Before scrolling

	A	B	C
1	Name	Midterm	Final
2	Ali	70	80
3	Ece	90	85
4	Can	60	72
5	Deniz	55	64

After scrolling down: row 1 stays fixed

	A	B	C
1	Name	Midterm	Final
41	Selin	88	91
42	Burak	76	70
43	Mert	65	81
44	Zehra	93	90

View → Freeze Panes → Freeze Top Row

Figure 3.2.6: Freezing panes. On the left the start of the list is visible. On the right it has been scrolled to rows 41–44, but the frozen row 1 (Name, Midterm, Final) stays on screen.

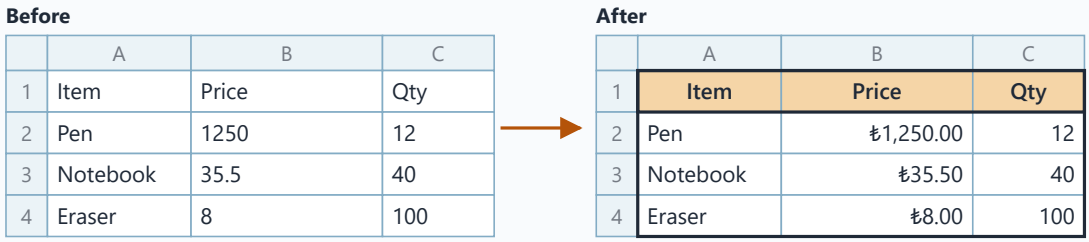
Freezing is not limited to the first row. The **Freeze First Column** command fixes column A; this is useful in wide tables whose row headers are in that column. To freeze both a row and a column, first select the first cell below and to the right of the part that is to stay fixed, then give the **Freeze Panes** command. For example, if B2 is selected, row 1 and column A freeze together.

3.2.3 Formatting

The same data reads very differently before and after formatting. A table with a clear header, aligned numbers, and explicit units is both easier to read and less likely to be misunderstood. Formatting in Excel has two separate layers: the appearance of the cell (font, color, border, alignment) and the way a number is displayed. The second layer matters separately for calculations.

3.2.3.1 Cell formatting

The cell-formatting commands are collected in the **Font**, **Alignment**, **Number**, and **Styles** groups on the Home tab. All the options are also in the **Format Cells** dialog, which opens with `Ctrl+1`. The tabs of the dialog are Number, Alignment, Font, Border, Fill, and Protection.



	A	B	C
1	Item	Price	Qty
2	Pen	1250	12
3	Notebook	35.5	40
4	Eraser	8	100

	A	B	C
1	**Item**	**Price**	**Qty**
2	Pen	৳1,250.00	12
3	Notebook	৳35.50	40
4	Eraser	৳8.00	100

Figure 3.2.7: The same table before and after formatting. On the right the header row is set apart with fill, prices are shown in currency format with two decimals, and borders have been added to the table.

- **Font and fill.** Bold text and a light fill color set the header row apart from the rest of the data.
- **Alignment.** Determines the horizontal and vertical position of the text inside the cell. Keeping numbers right-aligned makes the digits line up.
- **Border.** Shows the edges of the table. Adding a heavy border to every cell, however, clutters the table and makes it harder to read.
- **Wrap Text.** Splits a long text over several lines inside the cell without widening the column.
- **Merging.** The **Merge & Center** command makes several cells look like one. It can be used in a report title, but merged cells in a data range break sorting and filtering.
- **Cell style.** Ready-made sets of formatting. A format is defined once and applied consistently across the whole workbook.

Common mistake

Merging cells in the middle of a data table. For example, if the department name of three students in the same department is typed into a single merged cell, Excel sees the department of the second and third rows as empty. Sorting stops with the warning “To do this, all the merged cells need to be the same size”; a filter also finds only the first row. The department name must be typed separately in every row.

3.2.3.2 Number formats

The number in a cell can be displayed in different formats. The value 0.25 appears as 25% in percentage format, and the value 45292 as 1/1/2024 in date format. Changing the format does not change the number stored in the cell; it only determines how that number is written on screen.

Definition 3.2.5

A **number format** is a rule that determines how the value stored in a cell is shown on screen. A format does not change the value: calculations always use the underlying value shown in the formula bar.

Stored value	Format	Shown in the cell
0.25	Percentage	25%
1250	Currency	৳1,250.00
45292	Date	1/1/2024
0.5	Time	12:00
123456789	Scientific	1.23E+08

Table 3.2.2: How the same kinds of values look in different number formats.

Each row of Table 3.2.2 shows a rule. Percentage format shows the value multiplied by 100 with a percent sign appended; the cell still holds 0.25. Currency format adds a thousands separator and two decimal places. Why date and time values are numbers is explained in 3.2.8.1. In short, 1 day corresponds to the number 1; 0.5 is half a day, that is, 12 hours. In scientific format 1.23E+08 means 1.23×10^{8}.

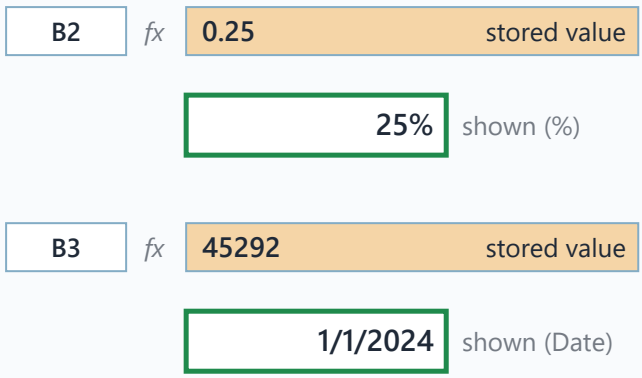


Figure 3.2.8: The formula bar shows the stored value, and the cell shows the formatted look. 0.25 is stored in B2 and 25% is shown; 45292 is stored in B3 and 1/1/2024 is shown.

The difference between the shown value and the stored value is most confusing with decimal places. The **Decrease Decimal** button shows a number with fewer digits but does not round it. Calculations are always done with the full value in the formula bar.

Example 3.2.4

The values 1.4, 1.4, and 1.4 are typed into cells A1, A2, and A3 in order, and all three are given a number format with 0 decimal places. The cells show 1. What does the formula `=SUM(A1:A3)` in A4 show?

1. **Stored values.** Because the format affects only the display, 1.4 stays in the cells.
2. **Total.** 1.4 + 1.4 + 1.4 = 4.2.
3. **Display.** If A4 also has a 0-decimal format, 4.2 is shown as 4.

A reader of the table expects 1 + 1 + 1 = 3 but sees 4. If you want calculations to use the displayed values, the value must actually be rounded: a formula such as `=ROUND(A1,0)` turns 1.4 into 1, and later calculations use this rounded value.

The ready-made number-format options are in the drop-down list in the Home → Number group: General, Number, Currency, Accounting, Short Date, Long Date, Time, Percentage, Fraction, Scientific, and Text. The **Custom** category in the Format Cells box lets you write your own format code. For example, the code `0.0 "kg"` shows the value 12 as “12.0 kg”. Because the number 12 is still stored in the cell, this value takes part in calculations.

Common mistake

Entering a percentage on the wrong scale. When 20 is typed into a cell already formatted as a percentage, Excel takes it as 20% and stores 0.2. If 20 is typed into an unformatted cell and the percentage format is applied afterwards, the cell shows 2000%, because the stored value is 20. The value of a ratio cell should always be checked in the formula bar: for 20% it should show 0.2 or 20%.

3.2.4 Formulas and operator precedence

The calculating power of Excel comes from formulas. A formula is what you would type into a calculator, enriched with cell addresses. This section covers the parts of a formula, the order in which Excel performs operations, and a surprising case that arises from that order.

3.2.4.1 The parts of a formula

Definition 3.2.6

A **formula** is a calculation expression typed into a cell that begins with an equals sign (=). A formula can contain four kinds of element: a **constant** (a value typed directly, such as 5), a **cell reference** (the address of another cell, such as C1), an **operator** (an operation sign such as +, −, *, /), and a **function** (a ready-made calculation such as SUM).

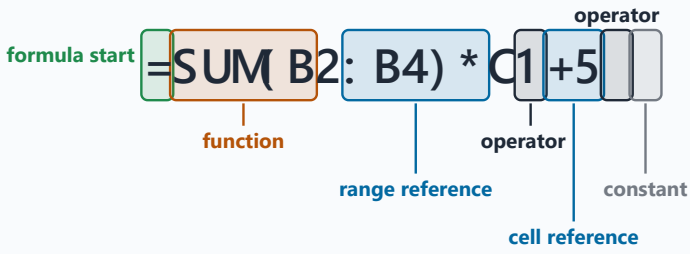


Figure 3.2.9: The parts of the formula `=SUM(B2:B4)*C1+5` : the equals sign at the start of the formula, the SUM function, the range reference B2:B4, the multiplication and addition operators, the cell reference C1, and the constant 5.

The formula in Figure 3.2.9 reads: “multiply the sum of B2, B3, and B4 by C1 and add 5 to the result.” If B2:B4 holds 10, 20, and 30 and C1 holds 2, the result is (10 + 20 + 30) × 2 + 5 = 60 × 2 + 5 = 125.

If the equals sign is forgotten, Excel stores what was typed as text, not as a formula. No calculation is done in a cell that holds `B2*C2` ; the text appears in the cell exactly as typed. The operators in Excel are as follows:

- **Arithmetic:** `+` addition, `-` subtraction and negation, `*` multiplication, `/` division, `^` exponentiation, `%` percent.
- **Comparison:** `=`, `>`, `<`, `>=`, `<=`, and `<>` for not equal. The result is TRUE or FALSE.
- **Concatenation:** `&` joins two pieces of text side by side. `=A1&" TL"` produces the text “1250 TL” if A1 holds 1250. Concatenation does not use the number's format; even if A1 is displayed in currency format, the result contains no thousands separator.
- **Reference:** a colon (`:`) forms a range: `B2:B4`.

Three principles guide good formula writing. First, when the order of operations is in doubt, use parentheses: `=(B2+C2)/2` is the average of two grades. Second, the result is recalculated when the referenced cells change; that is why formulas are written with addresses instead of values. Third, constants such as a VAT rate or a discount rate are not embedded in the formula but typed into a separate cell. When the rate changes, only that cell is updated.

3.2.4.2 Operator precedence

When a formula has several operators, Excel applies them in a particular order. This order largely resembles precedence in mathematics: first parentheses, then exponents, then multiplication and division, and last addition and subtraction. Excel's order has two additional levels: negation (the minus sign) and percent.

> **Definition 3.2.7**
>
> **Operator precedence** is the set of rules that determines the order in which the operators in a formula are applied. Operators of the same precedence are applied from left to right.

Precedence	Operation	Example
1	Parentheses	`=(A1+B1)*2`
2	Negation (sign)	`=-A1`
3	Percent	`=A1*10%`
4	Exponent	`=A1^2`
5	Multiplication and division	`=A1*B1/C1`
6	Addition and subtraction	`=A1+B1-C1`
7	Concatenation	`=A1&" TL"`
8	Comparison	`=A1>B1`

Table 3.2.3: Operator precedence in Excel (1 is highest).

The “left to right” rule applies to operations at the same level. In the formula `=12/3*2`, division and multiplication have the same precedence. Excel does the operation on the left first: 12 / 3 = 4, then 4 × 2 = **8**. If the multiplication were done first, 12 / 6 = 2 would result; that is wrong.

It also helps that comparison comes last. In the formula `=A1+B1>100`, the addition is done first and then the total is compared with 100. So the question "does the sum of the two grades exceed 100?" can be written without parentheses.

Example 3.2.5

What is the result of the formula `=5+3*2^2-8/4`? What is the result if parentheses are added at the start, giving `=(5+3)*2^2-8/4`?

1. **Exponent.** The formula has no parentheses or negation; the highest precedence is the exponent: 2^2 = 4. The formula becomes `5+3*4-8/4`.
2. **Multiplication and division, left to right.** 3 × 4 = 12, then 8 / 4 = 2. The formula becomes `5+12-2`.
3. **Addition and subtraction, left to right.** 5 + 12 = 17, 17 − 2 = **15**.
4. **The parenthesized form.** Parentheses first: 5 + 3 = 8. Then the exponent: 2^2 = 4. Then multiplication and division: 8 × 4 = 32 and 8 / 4 = 2. Last, 32 − 2 = **30**.

A single pair of parentheses doubled the result. Parentheses are the only way to change the order of operations.

3.2.4.3 Evaluating a formula step by step

When Excel evaluates a formula, it first replaces the cell references with the values in those cells. Then it carries out the operations one at a time in order of precedence. At each step the formula gets a little shorter, until a single value is left. This process can be followed step by step on screen with the **Formulas → Formula Auditing → Evaluate Formula** command. Table 3.2.4 shows the evaluation of the formula `=2+3*(B1-1)^2` when B1 holds 4.

Step	Operation performed	New form of the formula
0	Start	`=2+3*(B1-1)^2`
1	The reference is replaced by its value: B1 → 4	`=2+3*(4-1)^2`
2	Inside the parentheses: 4 − 1 = 3	`=2+3*3^2`
3	Exponent: 3^2 = 9	`=2+3*9`
4	Multiplication: 3 × 9 = 27	`=2+27`
5	Addition: 2 + 27 = 29	`=29`

Table 3.2.4: Evaluating the formula `=2+3*(B1-1)^2` (B1 = 4).

The result is **29**. Figure 3.2.10 shows the first step of this process. If the parentheses were removed and `=2+3*B1-1^2` were typed, the calculation would go quite differently: exponent 1^2 = 1, multiplication 3 × 4 = 12, addition and subtraction 2 + 12 − 1 = 13. Parentheses are the way to tell Excel "first take the difference, then square it".

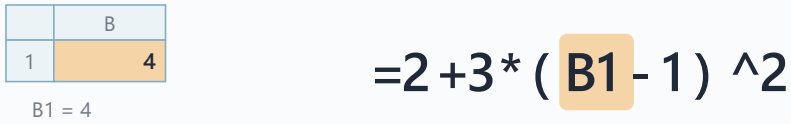


Figure 3.2.10: The first step of the evaluation. Excel replaces the reference B1 in the formula with the value in cell B1 (4); the operations are then done in order of precedence.

3.2.4.4 Negation and exponents

In Table 3.2.3 negation is second and exponentiation fourth. That is, in Excel the minus sign is applied before the exponent. In mathematics, however, the expression -2^2 is read as $-(2^2) = -4$; the exponent is taken first and the minus applied afterwards. This difference produces a surprising result in Excel.

Example 3.2.6

What do the formulas `=-2^2`, `=0-2^2`, and `=-(2^2)` give in Excel?

1. `=-2^2`. The leading minus is negation and comes before the exponent. First −2 is formed, then (−2)^2 = **4** is calculated.
2. `=0-2^2`. The minus here is not negation but subtraction; subtraction comes after the exponent. First 2^2 = 4, then 0 − 4 = **−4**.
3. `=-(2^2)`. Parentheses first: 2^2 = 4. Then negation: **−4**.

To get the mathematical result $-2^2 = -4$, the second or third form is used. The same applies to cell references: if A1 holds 3, the formula `=-A1^2` gives 9, not −9.

Another property of the exponent operator is that it is applied from left to right. Excel calculates the formula `=2^3^2` as (2^3)^2 = 8^2 = 64. In mathematics, by contrast, exponents are read from right to left: 2^(3^2) = 2^9 = 512. Writing parentheses in such expressions makes the reader and Excel understand the same thing.

Common mistake

Transferring an engineering formula to Excel without parentheses. For example, if the term e^{-x^2} that appears in the bell curve is written as `=EXP(-A1^2)`, Excel calculates $e^{(-x)^2} = e^{x^2}$. For A1 = 2 the expected value is about 0.018, but the result comes out about 54.6. The correct forms are `=EXP(-(A1^2))` or `=EXP(0-A1^2)`. In every exponent expression with a minus sign, the placement of the parentheses should be thought through, and the result compared with a hand-calculated value.

3.2.5 Cell references

A formula often does not stay in a single cell. If the same calculation is to be done for all the rows of a column, the formula is written once and copied down. The addresses in the copied formula change by themselves according to the new position. Knowing the rule of this change is perhaps the most important step in using Excel correctly. This section covers the three types of reference, how they behave when copied, and the most common reference error.

3.2.5.1 Types of reference

Definition 3.2.8

A **cell reference** is the address a formula writes in order to use the value of another cell. It has three types:

- A **relative reference**, `A1` : when the formula is copied, both the column letter and the row number shift as much as the formula shifts.
- An **absolute reference**, `$A$1` : points to the same cell wherever the formula is copied.
- A **mixed reference**, `$A1` or `A$1` : only the column or only the row is fixed; the other part shifts.

The meaning of the dollar sign is simple: `$` locks the part immediately after it. In `$A$1` there are two signs; both the column and the row are locked. In `$A1` there is a sign only before the letter A; the column is locked and the row is free. In `A$1` , the row is locked and the column is free. The dollar sign has nothing to do with currency; it is only a lock symbol.

References are not limited to a single cell on the same sheet. A range is written with two addresses (`A1:A10`), a cell on another sheet with the sheet name (`Sales!B2`), and a cell given a meaningful name with that name (`VAT_Rate`). These last two forms are covered in 3.2.5.8.

3.2.5.2 References when filling down

The fastest way to copy a formula is the **fill handle**. The handle is the small square at the bottom-right corner of the selected cell. When the mouse is brought to this square, the pointer turns into a thin plus sign. When the handle is dragged down, the formula is copied to the cells below; when it is dragged right, it is copied to the cells on the right.

Let us run a small experiment to see the difference between the reference types. Let cells A1:A4 hold 5, 10, 15, and 20. In B1 type `=A1` , in C1 `=$A$1` , and in D1 `=A$1` . All three formulas point to A1 and all three give 5. Then B1:D1 is selected and dragged down with the fill handle to row 4. Figure 3.2.11 shows the result.

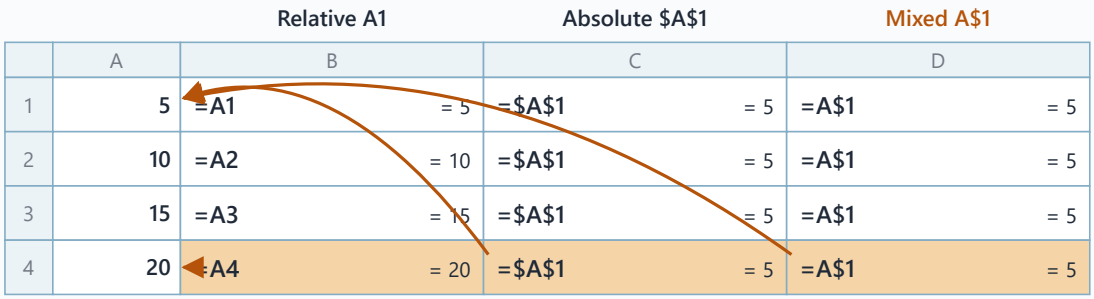


Figure 3.2.11: The three reference types after filling down. The relative `A1` shifts down one row at each row (A2, A3, A4) and gives 5, 10, 15, 20. `$A$1` and `A$1` keep the row locked, so they point to A1 in every row and always give 5.

Row	B (relative)	C (absolute)	D (mixed, row locked)
1	`=A1` → 5	`=$A$1` → 5	`=A$1` → 5
2	`=A2` → 10	`=$A$1` → 5	`=A$1` → 5
3	`=A3` → 15	`=$A$1` → 5	`=A$1` → 5
4	`=A4` → 20	`=$A$1` → 5	`=A$1` → 5

Table 3.2.5: The formulas row by row when filling down (A1:A4 = 5, 10, 15, 20).

The table shows an interesting result: when filled down, `A$1` behaves the same as `$A$1`. The reason is that in filling down, only the row number wants to shift. If the row is locked, the address does not change. The column lock has no effect, because the column does not shift anyway. The difference between the two references appears only when the formula is filled right.

3.2.5.3 References when filling right

This time let the data be in row 1: A1:D1 holds 5, 7, 9, and 11. Four reference types are typed into cells A3:A6: `=A1`, `=$A$1`, `=A$1`, and `=$A1`. All four give 5. Then A3:A6 is selected and filled right as far as column D.

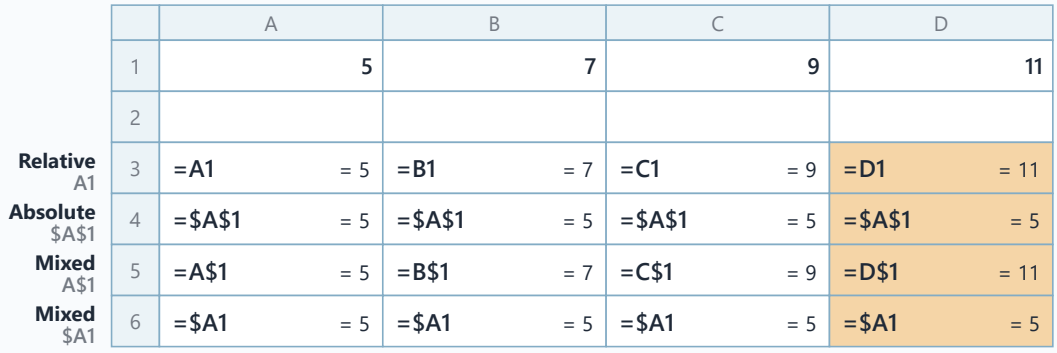


Figure 3.2.12: The four reference types after filling right. The relative reference and `A$1` shift along the

columns (B1, C1, D1) and give 7, 9, 11. `$A$1` and `$A1` keep the column locked, so they always point to A1.

In filling right, only the column letter wants to shift. In `A1` and `A$1` the column is free: they move to B1, C1, D1. In `$A$1` and `$A1` the column is locked: they always stay at A1. This time `$A1` behaves like an absolute reference. The rule that comes out of the two experiments is this:

- `$A$1` never shifts in any direction.
- `A$1` does not shift when filled down and shifts when filled right.
- `$A1` does not shift when filled right and shifts when filled down.
- `A1` shifts in both directions.

3.2.5.4 The copying rule

The fill handle carries a formula to adjacent cells. A formula can also be copied to a distant cell with `Ctrl+C` and `Ctrl+V`. The rule is the same: Excel finds the column and row difference between the source cell and the target cell and adds this difference to the unlocked parts of the formula.

Definition 3.2.9

The copying rule. Suppose a formula is copied to a cell with column difference Δc and row difference Δr. In each reference in the formula, a column letter without `$` in front of it shifts by Δc. A row number without `$` in front of it shifts by Δr. Parts marked with `$` do not change.

Example 3.2.7

Cell C3 holds the formula `=A1+$B$2+B$1+$A2`. The formula is copied to E7. What is the formula in E7?

1. **Difference.** From C to E is 2 columns right (Δc = 2), from 3 to 7 is 4 rows down (Δr = 4).
2. `A1`. Both parts are free: A + 2 = C, 1 + 4 = 5 → `C5`.
3. `$B$2`. Both parts are locked → `$B$2` stays as it is.
4. `B$1`. Column free, row locked: B + 2 = D, row 1 → `D$1`.
5. `$A2`. Column locked, row free: A stays, 2 + 4 = 6 → `$A6`.

Result: `=C5+$B$2+D$1+$A6`. Each reference is treated separately; no single rule is applied to the formula as a whole.

This rule works in the opposite direction too. When a formula is copied up or left, the differences are negative, and the addresses shift backward. If the shift runs off the sheet, for example if `=A1` is copied up one row, Excel cannot find a valid address and gives a `#REF!` error.

Cutting and pasting (`Ctrl+X`), by contrast, differs from copying. The references in a moved formula do not change; the formula keeps pointing to the same cells. The rule is only for copied formulas.

To try out the behavior of the reference types, you can use the Excel Cell References simulator on the course page. In the simulator, a formula is selected and dragged with the fill handle, and the formula produced in each cell and its result are seen at the same time.

3.2.5.5 Mixed reference: the multiplication table

The clearest use of a mixed reference is in tables filled in two directions at once. In a multiplication table each cell is the product of the number at the head of its row and the number at the head of its column. Let A2:A4 hold 4, 5, 6 and B1:D1 hold 1, 2, 3. Can a single formula typed into B2 fill the whole table?

The formula will have two references. The first points to the number at the head of the row. This number is always in column A, but its row must change with the cell: `$A2`. The second points to the number at the head of the column. This number is always in row 1, but its column must change with the cell: `B$1`. The formula is `=$A2*B$1`.

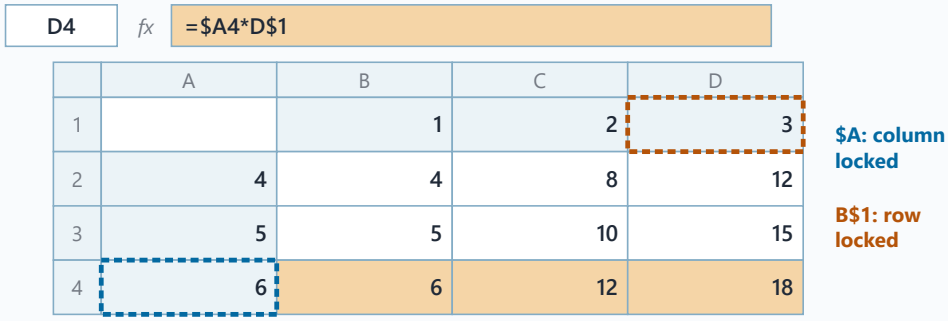


Figure 3.2.13: The formula `=$A2*B$1` typed in B2 has been filled into B2:D4. In D4, the last cell filled, the formula is `=$A4*D$1` and the result is 6 × 3 = 18. `$A` locks the column and `$1` locks the row.

Row	B	C	D
2	`=$A2*B$1` → 4	`=$A2*C$1` → 8	`=$A2*D$1` → 12
3	`=$A3*B$1` → 5	`=$A3*C$1` → 10	`=$A3*D$1` → 15
4	`=$A4*B$1` → 6	`=$A4*C$1` → 12	`=$A4*D$1` → 18

Table 3.2.6: The formula and result of each cell in the multiplication table.

Looking at Table 3.2.6 row by row makes the rule plain. Going right, only the column letter of B$1 changes (B → C → D). Going down, only the row number of $A2 changes (2 → 3 → 4). A single formula gives the right result in all nine cells.

Common mistake

Using a fully absolute reference such as `=$A$2*$B$1` in a multiplication table. When this formula is copied across the whole table, every cell gives 4 × 1 = 4. Conversely, a fully relative formula such as `=A2*B1` is wrong too: copied to C3 it becomes `=B3*C2`. Because B3 and C2 hold formulas of their own, the error compounds: B3 = 5 × 4 = 20, C2 = 4 × 2 = 8, and C3 = 20 × 8 = 160. The correct value is 10. The right reference type is found with the question: "In which direction must this value stay fixed?"

3.2.5.6 The F4 key

Instead of typing the dollar signs by hand, the F4 key can be used. While a formula is being edited and the cursor is on a reference, each press of F4 turns the reference type into the next one. The cycle completes in four presses and the reference returns to its first form. Table 3.2.7 shows the cycle for the reference E1 in the formula `=B2*E1`.

F4 press	Formula	Column (E)	Row (1)
Not pressed	`=B2*E1`	shifts	shifts
1.	`=B2*$E$1`	constant	constant
2.	`=B2*E$1`	shifts	constant
3.	`=B2*$E1`	constant	shifts
4.	`=B2*E1`	shifts	shifts

Table 3.2.7: Changing the reference type with the F4 key.

F4 works only while a formula is being edited and the cursor is inside a reference or right next to it. On many laptops the function keys are assigned to things like volume or brightness, so `Fn+F4` is needed. On a Mac the shortcut is different. The `$` sign can always be typed by hand too.

3.2.5.7 A relative-reference error in a copied formula

Using a relative reference in the wrong place leads to one of the sneakiest errors in Excel. It is sneaky because Excel gives no error message; since the results are numbers, they look right at first glance. Let us follow it with an example.

The prices of three items are in B2:B4: notebook 200, pen 150, eraser 80. The discount rate is kept in a single cell: E1 = 20%. The discounted-price formula will be typed into C2 and copied down. The first attempt is `=B2*(1-E1)`. In C2 the result is 200 × (1 − 0.2) = 200 × 0.8 = 160; this is correct. What happens when the formula is copied down?

	A	B	C	D	E
1	Item	Price	Discounted	Discount	20%
2	Notebook	200	160		
3	Pen	150	150		
4	Eraser	80			

=B3*(1-E2)

Figure 3.2.14: When `=B2*(1-E1)` in C2 is copied down, it becomes `=B3*(1-E2)` in C3. Because E2 is empty, no discount is applied and C3 shows the price itself (150).

Cell	Relative: formula	Result	Absolute: formula	Result
C2	`=B2*(1-E1)`	200 × 0.8 = 160	`=B2*(1-$E$1)`	200 × 0.8 = 160
C3	`=B3*(1-E2)`	150 × 1 = 150	`=B3*(1-$E$1)`	150 × 0.8 = 120
C4	`=B4*(1-E3)`	80 × 1 = 80	`=B4*(1-$E$1)`	80 × 0.8 = 64

Table 3.2.8: The two forms of the discount formula when copied down (E1 = 20%, E2 and E3 empty).

In the relative form both references have shifted one row. B2 becoming B3 is what we want, because each row must take its own price. E1 becoming E2, however, is the error. E2 is empty and Excel counts an empty cell as 0. The formula calculates 150 × (1 − 0) = 150; the discount is never applied. The same happens in C4.

The fix is to lock the rate cell. In the formula in C2 the cursor is placed on E1 and F4 is pressed once: `=B2*(1-$E$1)`. The price reference stays relative and the rate reference becomes absolute. When the formula is copied down, B3 and B4 shift but $E$1 stays in place. The results are 160, 120, and 64.

	A	B	C	D	E
1	Item	Price	Discounted	Discount	20%
2	Notebook	200	160		
3	Pen	150	120		
4	Eraser	80	64		

=B4*(1-$E$1)

Figure 3.2.15: The corrected formula. In C4, `=B4*(1-$E$1)` → 80 × 0.8 = 64. If the rate in E1 is changed to 30%, all three results update at once.

The absolute reference has one more benefit. If the rate is raised to 30%, only E1 is changed; the three results become 140, 105, and 56 at once. If the rate had been written into each formula separately, all three formulas would have had to be corrected one by one.

Example 3.2.8

In a stationery order, B2:B6 hold the quantity and C2:C6 the unit price (₺): pen 12 × 8; notebook 5 × 30; eraser 20 × 4.5; ruler 6 × 12.75; folder 10 × 22. The VAT rate is 20% in F1. The amount will go in column D and the amount with VAT in column E, and the formulas will

be copied down to row 6.

1. **Amount.** `=B2*C2` in D2. Both references must shift to their own row on every row; both stay relative. Results: 12 × 8 = 96; 5 × 30 = 150; 20 × 4.5 = 90; 6 × 12.75 = 76.5; 10 × 22 = 220.
2. **Amount with VAT.** `=D2*(1+$F$1)` in E2. The amount shifts with the row; the rate is fixed. Results: 96 × 1.2 = 115.2; 150 × 1.2 = 180; 90 × 1.2 = 108; 76.5 × 1.2 = 91.8; 220 × 1.2 = 264.
3. **Check.** The formula in E6 should be `=D6*(1+$F$1)`. If this is what the formula bar shows, the copying is correct.
4. **Totals.** `=SUM(D2:D6)` = 96 + 150 + 90 + 76.5 + 220 = 632.5 ₺. `=SUM(E2:E6)` = 759 ₺. The difference 759 − 632.5 = 126.5 ₺ is the VAT, which equals 632.5 × 0.2.

An easy way to catch errors of this kind is to make the formulas visible instead of the results. The **Formulas → Formula Auditing → Show Formulas** command, or the `Ctrl+`` shortcut, shows the formula itself in every cell in place of the result. If all the formulas in a column should point to a single rate cell, check whether that reference stays the same on every row. When the same command is given once more, the view returns to normal.

Common mistake

Checking only the first row of a copied formula. A relative-reference error does not show at all in the first row; the result in C2 is 160 in both forms. The error appears only in the second and later rows. After a formula is copied, the formula of at least one middle row and of the last row should be opened and looked at too.

3.2.5.8 References to another sheet and named ranges

In large workbooks the data, calculation, and report are often on separate sheets. If a formula is to use a cell on another sheet, the sheet name is written in front of the address with an exclamation mark between them: `Sales!B2` is cell B2 of the Sales sheet. If the sheet name contains a space, the name is put in single quotes: `'Sales 2024'!B2`. While writing a formula, clicking the other sheet's tab and selecting the cell makes Excel itself write this address.

Definition 3.2.10

A **named range** is a meaningful name given to a cell or a range. The name is used in a formula in place of the address and always points to the same cell; that is, it behaves like an absolute reference.

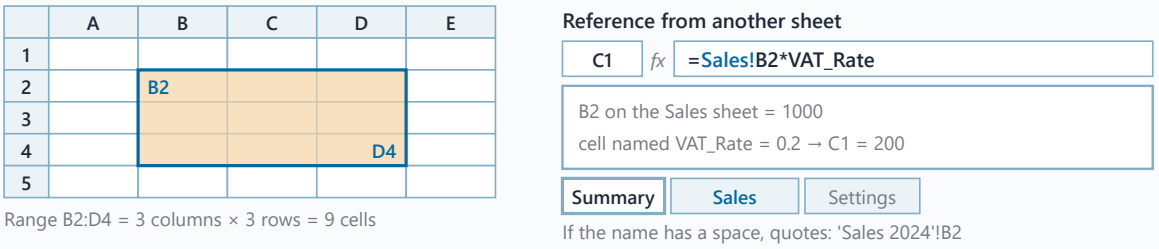


Figure 3.2.16: On the left the range B2:D4: 3 columns × 3 rows = 9 cells. On the right, cell C1 on the Summary sheet multiplies cell B2 of the Sales sheet (1000) by the cell named VAT_Rate (0.2) and gives 200.

The shortest way to name a cell is to select the cell, type the name in the Name Box, and press Enter. The **Formulas → Defined Names → Define Name** command does the same job; the **Name Manager** lists all the names in the workbook. Spaces cannot be used in names, so words are joined with an underscore: `VAT_Rate`. A name must start with a letter or an underscore and must not look like a cell address; for example `VAT1` is not accepted because it looks like a column name and a row number.

The formula in Figure 3.2.16 is `=Sales!B2*VAT_Rate`. Because B2 on the Sales sheet holds 1000 and the cell named VAT_Rate holds 0.2, the result is 1000 × 0.2 = 200. `=Sales!B2*Settings!$B$1` gives the same result, but the reader sees at once what is multiplied by what. Because a name behaves like an absolute reference, VAT_Rate stays in place when the formula is copied.

3.2.6 Basic and logical functions

Typing `=B2+B3+…+B101` to add up a hundred numbers in a column is both tiring and error-prone. Excel has ready-made functions for frequent calculations like this. This section covers the structure of a function, the five most-used functions, and the logical functions used for making decisions.

3.2.6.1 The structure of a function and basic functions

Definition 3.2.11

A **function** is a ready-made formula that does a particular calculation and is called by its name. After the function name a parenthesis is opened, and the values that go into the calculation, the **arguments**, are written inside the parentheses. In English-language Excel the arguments are separated by commas: `=IF(B2>=60,"Pass","Fail")`.

On a computer set to Turkish (and in several other regional settings), Excel uses localized function names and separates arguments with semicolons, because the comma is the decimal separator there: the same formula is written `=EĞER(B2>=60;"Pass";"Fail")`, and `=TOPLA(1,5;2)` adds 1,5 and 2 to give 3,5. When a file is opened in another language of Excel, function names and separators are

translated automatically; inside the file the formula is stored independently of the language. A formula example found on the internet, however, must be adapted by hand for an Excel of a different language.

Function	Example	What it does
SUM	`=SUM(B2:B10)`	Adds the numbers in the range.
AVERAGE	`=AVERAGE(B2:B10)`	Gives the arithmetic mean of the numbers.
MAX	`=MAX(B2:B10)`	Gives the largest number.
MIN	`=MIN(B2:B10)`	Gives the smallest number.
COUNT	`=COUNT(B2:B10)`	Counts the cells that contain numbers.
COUNTA	`=COUNTA(B2:B10)`	Counts all cells that are not empty.
COUNTIF / SUMIF	`=COUNTIF(B2:B10,">=50")`	Counts or adds the cells that meet a condition.
TODAY	`=TODAY()`	Gives today's date.

Table 3.2.9: Frequently used basic functions.

A function can be written in three ways. The first is to type its name directly: when `=SU` is typed, Excel lists the functions that start with SU, and the Tab key completes the selected one. The second is to choose the function from a list with **Formulas → Insert Function**; the box that opens explains what each argument is. The third is the **Home → AutoSum** button. This button finds the numbers above the selected cell and writes the `=SUM(…)` formula itself. Its shortcut is `Alt+=`.

3.2.6.2 Functions on the same data

How the basic functions behave toward text and empty cells matters for interpreting the results correctly. Let the range B2:B7 hold 12, 7, the text “absent”, 20, an empty cell, and 5. When the five functions are applied to the same range, Figure 3.2.17 results.

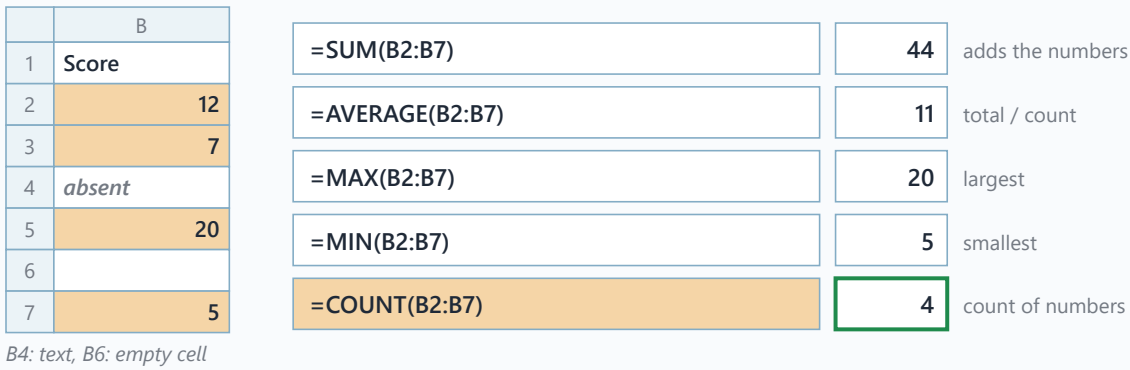


Figure 3.2.17: Five basic functions on the same range. The text “absent” in B4 and the empty cell in B6 are ignored by all the functions: SUM 44, AVERAGE 11, MAX 20, MIN 5, COUNT 4.

Example 3.2.9

With B2:B7 = 12, 7, “absent”, 20, empty, 5, let us find the results of the five functions step by step.

1. **Which cells are numbers?** 12, 7, 20, and 5. “absent” is text and B6 is empty; neither enters the calculation.
2. **SUM.** 12 + 7 + 20 + 5 = **44**.
3. **AVERAGE.** The total is divided by the count of numbers: 44 / 4 = **11**. The divisor is 4, not 6; if the empty cell were counted as 0, the divisor would be 5 and the result would be 8.8.
4. **MAX and MIN.** The largest number is **20** and the smallest is **5**.
5. **COUNT.** The number of cells containing numbers: **4**. COUNTA, which counts all cells that are not empty, also counts the text and gives **5**.

The difference between an empty cell and a cell containing 0 is decisive in an average. If a student did not take the exam, the cell is left empty and does not enter the average. If the student took the exam and got 0, 0 is typed into the cell and pulls the average down. In the two cases the table may look the same, but the class average comes out different.

Common mistake

Mistaking COUNT for “number of filled cells”. COUNT counts only numbers; if COUNT is used to find the number of students in a list of names, the result is 0. For the number of records in a column with text, COUNTA is used.

3.2.6.3 The IF function

In some calculations the result depends on a condition: if the grade is 60 or above, “Pass” is written, otherwise “Fail”. Decisions like this are made with the IF function.

Definition 3.2.12

The **IF** function takes three arguments: `=IF(logical_test, value_if_true, value_if_false)`. A **logical test** is an expression whose result is TRUE or FALSE. If the test is TRUE, the second argument is returned; if FALSE, the third.

```
=IF(B2>=60, "Pass", "Fail")
```

In this formula the test is `B2>=60`. If it is true, the text “Pass” is returned; if false, the text “Fail”. Text is written inside double quotes in formulas. If the quotes are forgotten, Excel takes the word `Pass` for a name and gives a `#NAME?` error. Figure 3.2.18 shows the decision structure of the function as a flowchart.

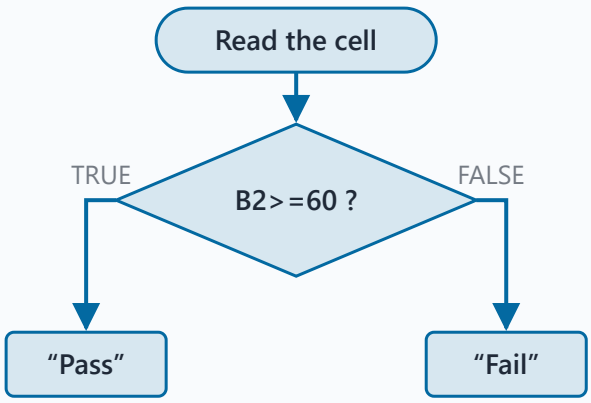


Figure 3.2.18: The flowchart of the IF function. The cell is read and `B2>=60` is tested; if the test is true, “Pass” is returned, otherwise “Fail”.

When the formula is typed into C2 and copied down, each row tests its own score. Let A2:B4 hold Ali 75, Can 45, and Ece 60. Table 3.2.10 shows the evaluation of each row.

Cell	Score read	Test	Result
C2	B2 = 75	75 >= 60 → TRUE	Pass
C3	B3 = 45	45 >= 60 → FALSE	Fail
C4	B4 = 60	60 >= 60 → TRUE	Pass

Table 3.2.10: Row-by-row evaluation of the formula `=IF(B2>=60,"Pass","Fail")`.

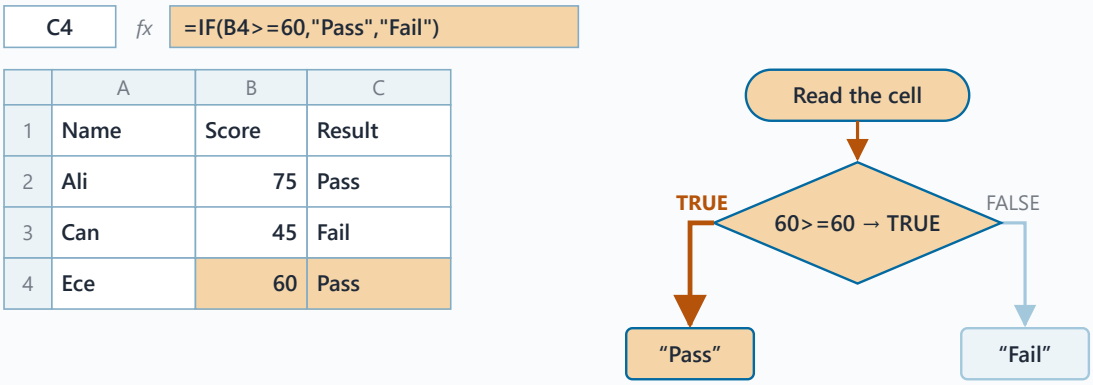


Figure 3.2.19: The evaluation of the last row. In C4, B4 = 60 is read; because 60 >= 60 is true, the second argument (“Pass”) is returned.

The third row shows how much the choice of comparison operator matters. `>=` includes equality; Ece, who scored 60, passes. Had `>` been typed in the formula, the test would be 60 > 60 → FALSE and Ece would show "Fail". Scores at the pass boundary are the best test values for checking whether a formula is written correctly.

The values IF returns need not be text. A number or another formula can be returned as well. For example, the formula `=IF(B2>=1000,B2*0.1,0)` gives the 10% discount amount on purchases of 1000 ₺ and above, and 0 below that. If B2 = 1250 the result is 125; if B2 = 800 it is 0.

3.2.6.4 AND, OR, NOT, and nested IF

Some decisions depend on several conditions. A rule such as "a student passes if the grade is at least 60 *and* attendance is at least 70%" cannot be written with a single comparison. Excel has three logical functions for combining conditions:

- **AND**: gives TRUE if all the conditions are TRUE, and FALSE if at least one is FALSE.
- **OR**: gives TRUE if at least one condition is TRUE, and FALSE if all are FALSE.
- **NOT**: reverses a single condition; it turns TRUE into FALSE and FALSE into TRUE.

Let a student's grade be B2 = 72, attendance C2 = 65 (as a percentage), and D2 hold 0. Table 3.2.11 shows the results of various formulas.

Formula	Result	Description
`=AND(B2>=60,C2>=70)`	FALSE	72 >= 60 true, 65 >= 70 false; the two are not both true.
`=OR(B2>=60,C2>=70)`	TRUE	At least one is true.
`=NOT(B2>=60)`	FALSE	TRUE is reversed.
`=IFERROR(B2/D2,"Cannot calculate")`	Cannot calculate	72 / 0 gives an error; text is shown instead.
`=IF(B2>=85,"AA",IF(B2>=70,"BB","CC"))`	BB	Nested IF: the value of the first true condition is returned.
`=IF(AND(B2>=60,C2>=70),"Pass","Fail")`	Fail	AND is used as the test of IF.

Table 3.2.11: Logical functions (B2 = 72, C2 = 65, D2 = 0).

The last row is the most frequently used pattern: AND or OR is used not to give a result on its own but as the test of IF. The student's grade is sufficient but attendance is insufficient; because AND gives FALSE, IF returns the third argument.

When more than two outcomes are needed, IFs are nested. A second IF is placed in the value_if_false part of an IF. Excel tests the conditions from left to right and returns the value of the first true condition; it never looks at the later conditions.

Example 3.2.10

What does the formula `=IF(B2>=85,"AA",IF(B2>=70,"BB","CC"))` return for B2 = 72, 90, and 69?

1. **B2 = 72.** Outer IF: 72 >= 85 → FALSE; move to the third argument, that is, the inner IF. Inner IF: 72 >= 70 → TRUE → **BB**.
2. **B2 = 90.** Outer IF: 90 >= 85 → TRUE → **AA**. The inner IF is never evaluated.
3. **B2 = 69.** 69 >= 85 → FALSE; 69 >= 70 → FALSE → **CC**.

Parentheses are checked by counting: each IF opens a parenthesis and closes its own. Since the formula has two IFs, there are two closing parentheses at the end.

Common mistake

Starting the conditions from the low threshold in a nested IF. In the formula `=IF(B2>=70,"BB",IF(B2>=85,"AA","CC"))` a student with 90 also passes the first test and gets "BB"; the "AA" condition is never reached. Conditions must be written starting from the highest threshold and going down.

With many tiers a nested IF grows long and becomes hard to read. In Microsoft 365 there is also the `IFS` function for this job: `=IFS(B2>=85,"AA",B2>=70,"BB",TRUE,"CC")`. This function tries test–value pairs in order. The `TRUE` test in the last pair gives the value used if none of the earlier conditions holds. Another way to handle tiered thresholds is the approximate-match lookup described in 3.2.7.4.

3.2.6.5 IFERROR

Some formulas give an error with certain inputs. In a division the denominator may be 0, or a lookup may be searching for a value that is not in the list. The IFERROR function is used to show a meaningful result instead of the error value in such cases.

Definition 3.2.13

The **IFERROR** function takes two arguments: `=IFERROR(value, value_if_error)`. If the first argument does not produce an error, its result is returned; if it does, the second argument is returned.

In the example in Table 3.2.11, `B2/D2` = 72 / 0 cannot be calculated and gives `#DIV/0!`. IFERROR catches this error and shows the text "Cannot calculate". If D2 held 4, 72 / 4 = 18 would not give an error; in that case IFERROR would return 18 as it is.

IFERROR hides the error but does not remove its cause. Why the denominator is 0 must still be investigated. Besides, this function catches all kinds of errors. A `#NAME?` error caused by a typo is also shown as “Cannot calculate”, and the real problem is overlooked. That is why IFERROR is used where an error is expected and its meaning is known.

3.2.7 Conditional counting, summing, and lookup

In real tables most questions contain a condition. “How many students are there in the Electrical department?”, “What is the total of these students' scores?”, “What is Ece's score?” This section covers the conditional counting and summing functions that answer such questions, and the lookup functions that fetch a value from a table. The same example list is used throughout the section: six students with Name, Dept, and Score columns. Ali (Electrical, 72), Ece (Computer, 91), Can (Electrical, 85), Deniz (Computer, 64), Burak (Mechanical, 78), and Selin (Electrical, 58). The list is in A1:C7; row 1 is the header.

3.2.7.1 COUNTIF and SUMIF

Definition 3.2.14

COUNTIF counts the cells in a range that meet a particular **criterion**: `=COUNTIF(range, criteria)`. **SUMIF** adds up the values in the rows that meet the criterion: `=SUMIF(criteria_range, criteria, sum_range)`. In SUMIF the criterion is tested in one column and the values to be summed are taken from another column in the same rows.

	A	B	C
1	Name	Dept	Score
2	Ali	Electrical	72
3	Ece	Computer	91
4	Can	Electrical	85
5	Deniz	Computer	64
6	Burak	Mechanical	78
7	Selin	Electrical	58

`=COUNTIF(B2:B7,"Electrical")`
counts the rows where Dept = “Electrical” = 3

`=SUMIF(B2:B7,"Electrical",C2:C7)`
adds the scores of the rows that meet the condition = 215

`=COUNTIF(C2:C7,">=80")`
counts the scores of 80 and above = 2

Figure 3.2.20: Three conditional formulas on the student list. “Electrical” appears 3 times in the Dept column; the total of the Electrical students' scores is 215; there are 2 students with a score of 80 or above.

Example 3.2.11

How does the formula `=SUMIF(B2:B7,"Electrical",C2:C7)` move through the rows?

1. **Row 2.** B2 = Electrical → meets the criterion; C2 = 72 is added to the total. Running total 72.
2. **Row 3.** B3 = Computer → does not match; C3 is skipped.
3. **Row 4.** B4 = Electrical → matches; 72 + 85 = 157.
4. **Rows 5 and 6.** Computer and Mechanical → both skipped.
5. **Row 7.** B7 = Electrical → matches; 157 + 58 = **215**.

With the same criterion, `=COUNTIF(B2:B7,"Electrical")` gives 3. The ratio of the two is the average of the Electrical students: 215 / 3 ≈ 71.7. This average can also be found directly with `=AVERAGEIF(B2:B7,"Electrical",C2:C7)`.

In SUMIF, the two ranges must be the same size and in the same rows. If the criterion is tested in B2:B7, the sum range must also be C2:C7; if C3:C8 is typed, each criterion is matched with the score one row below, and the result is silently wrong. If the sum range is omitted, the criterion range itself is summed. `=SUMIF(C2:C7,">=70")` adds up the scores of 70 and above: 72 + 91 + 85 + 78 = 326.

3.2.7.2 Writing criteria and multi-condition functions

A criterion can be text, a number, or a comparison. Comparison criteria are written inside double quotes. Table 3.2.12 shows the commonly used forms together with the results on the example list.

Formula	Meaning	Result
`=COUNTIF(B2:B7,"Electrical")`	Equal to Electrical	3
`=COUNTIF(B2:B7,"<>Electrical")`	Anything other than Electrical	3
`=COUNTIF(C2:C7,">=80")`	Scores of 80 and above	2
`=COUNTIF(C2:C7,"<60")`	Scores below 60	1
`=COUNTIF(B2:B7,"E*")`	Departments that start with E (asterisk: any string)	3
`=COUNTIF(C2:C7,">="&F1)`	The threshold is taken from cell F1 (if F1 = 80)	2

Table 3.2.12: Writing criteria, and the results on the example list.

The syntax in the last row matters. If the threshold is in a cell, the comparison sign is written inside quotes and joined to the cell address with the `&` operator. If `">=F1"` is typed, Excel looks for the word “F1” and matches no number. Criteria are not case-sensitive: “electrical” and “Electrical” count as the same.

When several conditions are searched at once, multi-condition functions are used. `COUNTIFS` takes range–criterion pairs and counts the rows that meet all of them at once. In `SUMIFS` the sum range is written first, followed by the range–criterion pairs.

Example 3.2.12

How many students in the Electrical department have a score of 70 or above? What is the total of these students' scores?

1. **Formulas.** `=COUNTIFS(B2:B7,"Electrical",C2:C7,">=70")` and `=SUMIFS(C2:C7,B2:B7,"Electrical",C2:C7,">=70")` .
2. **First condition.** Electrical rows: Ali 72, Can 85, Selin 58.
3. **Second condition.** Of these, 70 and above: Ali 72 and Can 85. Selin is eliminated.
4. **Results.** Count **2**, total 72 + 85 = **157**.

In multi-condition functions the conditions are joined by AND: a row must meet all the conditions to be counted.

3.2.7.3 Exact match with VLOOKUP

Fetching the information of a particular record from a table is a common job: finding a name from a student number, a price from a product code, or a score from a student's name. VLOOKUP solves this by searching in the first column of the table.

Definition 3.2.15

VLOOKUP takes four arguments: `=VLOOKUP(lookup_value, table_array, col_index_num, range_lookup)` . It searches for the lookup value from top to bottom in the *first column* of the table; in the row where it finds it, it returns the value in the table's `col_index_num` -th column. If the last argument is 0 (FALSE), an **exact match** is made; if it is 1 (TRUE), an **approximate match**.

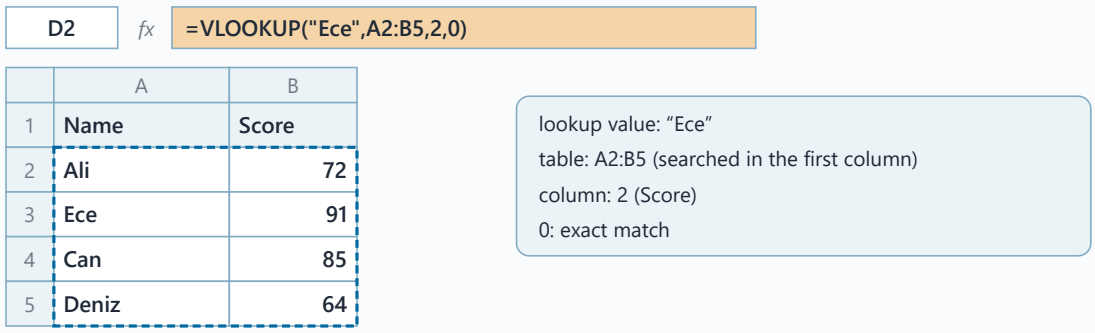


Figure 3.2.21: The parts of the formula `=VLOOKUP("Ece",A2:B5,2,0)` : the lookup value "Ece", the table A2:B5 (the search is made in the first column), column number 2 (Score), and 0 for an exact match.

Example 3.2.13

A2:B5 hold Ali 72, Ece 91, Can 85, Deniz 64. What do `=VLOOKUP("Ece",A2:B5,2,0)` and `=VLOOKUP("Mert",A2:B5,2,0)` return?

1. **A2.** "Ali" ≠ "Ece" → move to the next row.
2. **A3.** "Ece" = "Ece" → found; the search stops.
3. **Result.** The row found is the 2nd row of the table. Because the 2nd column of the table is B, B3 is returned: **91**.
4. **"Mert".** None of A2, A3, A4, and A5 is "Mert". After all the rows have been tried, the result is **#N/A** (value not found).

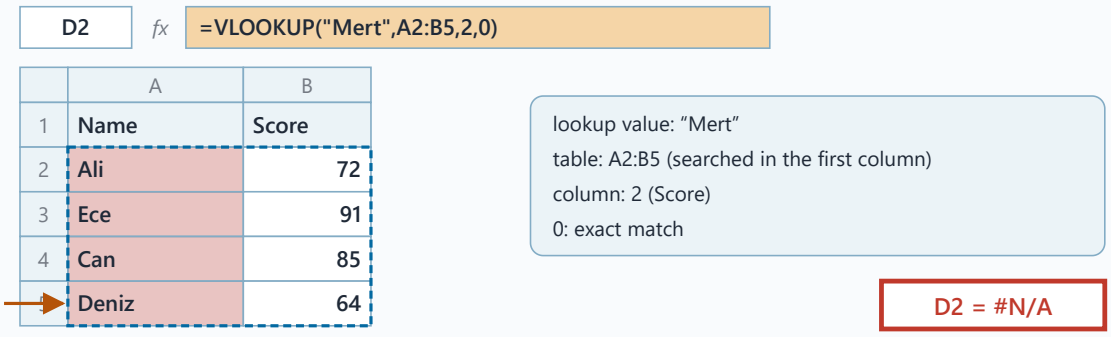


Figure 3.2.22: "Mert" is not found in any row of the first column; VLOOKUP with an exact match returns #N/A.

The column number is counted not by the sheet's column letter but by the table's own columns. If the table is C5:F20, C is the 1st, D the 2nd, and E the 3rd column. Because this number is typed into the formula as a constant, when a new column is inserted in the middle of the table the number points to the wrong column. Another limit of VLOOKUP is that it can look only to the right. The searched column is always the first column of the table; a value to the left of the first column cannot be fetched.

Common mistake

Leaving out the fourth argument. If this argument is omitted, Excel does an approximate match. In an unsorted list of names, an approximate match can return a wrong person's score without any error. In searches for names, codes, or numbers, the last argument should always be 0.

In an exact match too, the lookup value must be typed exactly. A trailing invisible space ("Ece ") breaks the match and the result is #N/A. Capitalization makes no difference: a search for "ece" also returns 91. If you want to show a message instead of an error for values that are not in the list, `IFNA`

is used: `=IFNA(VLOOKUP(D2,A2:B5,2,0),"No record")`. IFNA catches only the #N/A error; because it does not hide other errors, it is safer than IFERROR.

3.2.7.4 Approximate match with VLOOKUP

The right use for approximate match is where a value falls into a range: a grading scale, a tax bracket, a shipping-rate table. In these tables the lookup value is often not found exactly in the table; what is found is which bracket it falls into.

Let D2:E6 hold a grading scale. In the first column the lower thresholds are sorted from smallest to largest: 0 → F, 50 → D, 60 → C, 70 → B, 85 → A. The letter for a score is found with this formula:

```
=VLOOKUP(B2, $D$2:$E$6, 2, TRUE)
```

The scale is locked with $, because when the formula is copied down every row must look at the same scale. The rule of approximate match is this: in the first column, *the largest threshold that is less than or equal to the lookup value* is found, and that row's value is returned.

Example 3.2.14

What are the letter grades for scores 72, 85, 49, and −5?

1. **72.** The thresholds 0, 50, 60, and 70 are less than or equal to 72; 85 is not. The largest suitable threshold is 70 → **B**.
2. **85.** Equality counts too: 85 ≤ 85. The largest suitable threshold is 85 → **A**.
3. **49.** Only 0 qualifies, because 50 > 49 → **F**.
4. **−5.** No threshold is less than or equal to −5 → **#N/A**.

Had an exact match been used, all scores not in the table, such as 72 and 49, would give #N/A. What the −5 result teaches is this: the smallest threshold in the scale must be the smallest possible value (here 0). Invalid scores can be prevented from the start with data validation, described in 3.2.11.4.

	A	B	C	D	E
1	Name	Score	Letter	Threshold	Letter
2	Ece	72	B	0	F
3	Can	85		50	D
4	Mert	49		60	C
5	Ada	-5		70	B
6				85	A

0 ≤ 72 ? yes
50 ≤ 72 ? yes
60 ≤ 72 ? yes
70 ≤ 72 ? yes
85 ≤ 72 ? no

Figure 3.2.23: Looking up a score of 72 on the scale. The thresholds 0, 50, 60, and 70 are less than or equal to 72, and 85 is not; the largest suitable threshold is 70, and the letter is B.

In an approximate match Excel does not scan the table row by row; it does a **binary search**. It looks at the value in the middle of the table and, if the lookup value is smaller, continues in the upper half, otherwise in the lower half. This method is very fast on large tables, but it depends on the first column being sorted from smallest to largest. On an unsorted scale Excel gives no error; it may return a wrong letter. That is why the scale is sorted before an approximate match is used.

3.2.7.5 XLOOKUP and FILTER

Microsoft 365 and Excel 2021 have new functions that remove the limits of VLOOKUP. These functions do not work in Excel 2019 and earlier versions, so the version used by the people who will open the file matters.

Definition 3.2.16

XLOOKUP takes the array to search and the array to return separately: `=XLOOKUP(lookup_value, lookup_array, return_array)`. By default it does an exact match. **FILTER** returns all the rows of a range that meet a condition: `=FILTER(array, include)`.

```
=XLOOKUP("Ece",A2:A5,B2:B5)      → 91
=FILTER(A2:B5,B2:B5>=80)         → Ece 91 / Can 85
```

XLOOKUP has no column number; the column to be returned is indicated directly. The formula does not break when a column is added to the table. The lookup array may also be to the right of the return array: if the names are in column B and the scores in column A, `=XLOOKUP("Ece",B2:B5,A2:A5)` still gives 91. VLOOKUP cannot make this lookup. If the lookup value is not found, XLOOKUP also gives #N/A; but a message can be written as the fourth argument: `=XLOOKUP("Mert",A2:A5,B2:B5,"No record")`.

FILTER, on the other hand, returns not a single value but a table of results. The result spreads over as many rows and columns as needed, starting from the cell where the formula is typed. This behavior is called **spilling**.

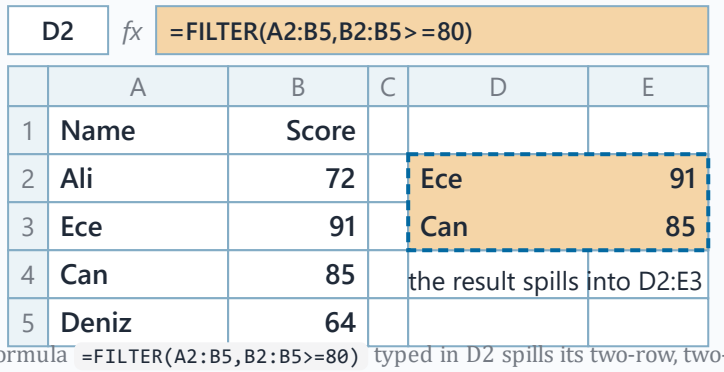


Figure 3.2.24: The formula `=FILTER(A2:B5,B2:B5>=80)` typed in D2 spills its two-row, two-column result into D2:E3: Ece 91 and Can 85.

If there is a filled cell in the way of a spilled result, Excel cannot place the result and gives a `#SPILL!` error. When the cell in the way is emptied, the result settles into place by itself. Only the first cell of the spilled range holds the formula; the other cells show the same formula in faded color in the formula bar and cannot be edited individually. When the source data changes, the spilled result updates too: if Deniz's score rises from 64 to 82, the FILTER result grows to three rows.

Common mistake

Typing other data below a spilled result. FILTER may be returning two rows today, but when the data changes it can grow to five. A total row typed right below it then blocks the result's path, and the whole result becomes #SPILL!. Enough empty space must be left below and to the right of spilling formulas.

3.2.8 Dates, times, and formula errors

Date and time calculations look confusing at first glance, but once you know how Excel stores them they turn into simple arithmetic. Formula errors are not random either; each error value reports a particular cause. This section first covers how dates and times are stored as numbers, then the error values and ways to fix them.

3.2.8.1 Date and time

Definition 3.2.17

Excel stores every date as a **date serial number**: the date 1/1/1900 is 1 and each following day is one more. A time is stored as a fraction of a day: 0.5 is half a day, that is, 12:00.



Figure 3.2.25: Dates are stored as serial numbers. 3/1/2024 = 45352, 3/15/2024 = 45366; their difference is found with `=B2-A2` as 14 days. A time is a fraction of a day: 0.5 → 12:00.

The benefit of this representation is clear: because dates are numbers, they can be calculated with arithmetic. The difference of two dates gives the number of days between them. Adding a number to a date gives a later date: 3/15/2024 + 30 = 4/14/2024. When a date cell is changed to General format, the stored serial number appears; this is an easy way to tell whether a cell really holds a date or text.

Example 3.2.15

A2 holds 2/1/2024 and B2 holds 3/1/2024. What does `=B2-A2` give? What would the result have been if the same month range were in 2023?

1. **Serial numbers.** 2/1/2024 = 45323, 3/1/2024 = 45352.
2. **Difference.** 45352 − 45323 = **29** days. 2024 is a leap year; February has 29 days.
3. **2023.** 2023 is not a leap year; the same difference is **28** days.

The result cell sometimes takes on a date format by itself, and instead of 29 a meaningless date such as 1/29/1900 appears. In that case the cell's number format is simply set to General or Number; the calculation is correct.

The main date and time functions and rules are these:

- `=TODAY()` gives today's date and `=NOW()` gives the date and time. Both update at every recalculation. In tracking delays, the formula `=TODAY()-C2` finds the days elapsed since the due date anew every day.
- `=DATE(2024,3,15)` builds a date from a year, a month, and a day. It is used to combine year, month, and day information that sits in separate columns.

- A date stored as text (if "3/15/2024" sits left-aligned) is not counted in calculations; subtraction may give a `#VALUE!` error. Such data usually arises when it is imported from another system.
- Times, being fractions of a day, can be added too. The duration between 09:00 and 11:45 is found with `=B2-A2` and shows as 02:45; this is 2.75 hours or 165 minutes.

Example 3.2.16

A lab assistant worked 9 hours a day for five days. The daily durations are typed as 09:00 in B2:B6. What does `=SUM(B2:B6)` show? If the hourly pay is 150 ₺, how is the weekly pay calculated?

1. **Total.** 5 × 9 = 45 hours. The stored value is 45 / 24 = 1.875 days.
2. **The standard time format.** The `h:mm` format drops whole days and shows only the remaining hours: 45 hours = 1 day + 21 hours → **21:00** appears in the cell.
3. **The right format.** The Custom format code is set to `[h]:mm`. The square brackets stop the hours from resetting at 24: **45:00**.
4. **Pay.** The stored value is in days. To convert it to hours, multiply by 24: `=B7*24*150` = 1.875 × 24 × 150 = **6750 ₺**.

Multiplying the time total directly by 150 (1.875 × 150 = 281.25) gives a wrong result. A duration stored in days must first be converted to hours.

3.2.8.2 Error values

Definition 3.2.18

An **error value** is a special result beginning with `#` that appears in a cell when a formula cannot be calculated. Each error value points to a different kind of problem.

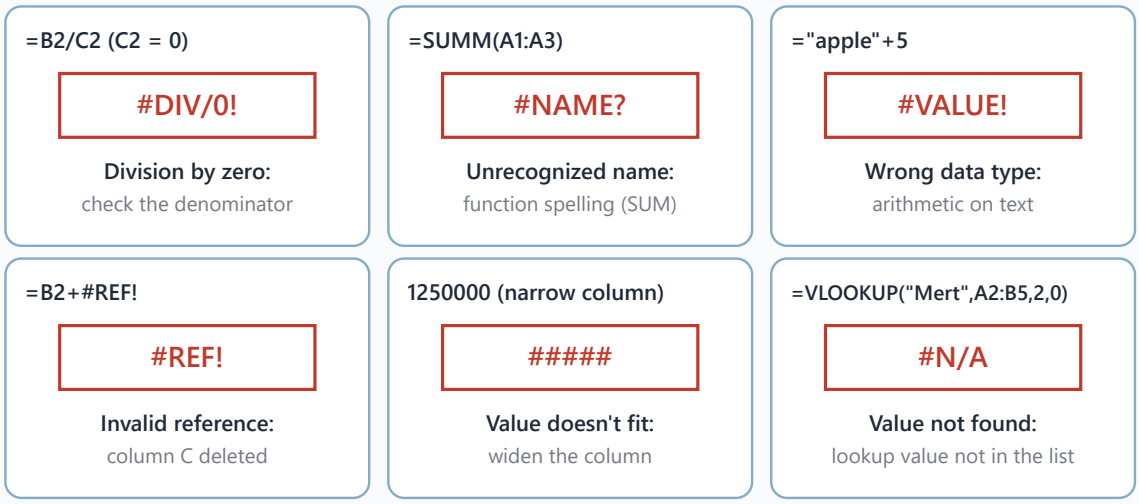


Figure 3.2.26: Frequently seen error values and their causes: division by zero (#DIV/0!), unrecognized name (#NAME?), wrong data type (#VALUE!), invalid reference (#REF!), a value that does not fit the column (#####), and a value not found (#N/A).

Error	Example	Cause	Fix
`#DIV/0!`	`=B2/C2`, C2 = 0	Division by zero; an empty cell also counts as 0.	Check the denominator, or `=IF(C2=0,"",B2/C2)`.
`#NAME?`	`=SUMM(B2:B4)`	Excel does not recognize the name: a typo or text without quotes.	`=SUM(B2:B4)`; text goes inside quotes.
`#VALUE!`	`=B2*2`, B2 = "12 TL"	Arithmetic on text.	Only 12 in the cell; the unit is shown with a number format.
`#REF!`	`=B2+#REF!`	The cell or column the formula used has been deleted.	Undo with Ctrl+Z, or retype the reference.
`#N/A`	`=VLOOKUP("Selin",A2:B4,2,0)`	The lookup value is not in the list.	Check the spelling, add the record, or use IFNA.
`#####`	1250000 in a narrow column	The value does not fit in the column; not a real error.	The column is widened.
`#SPILL!`	`=FILTER(…)`	There is a filled cell in the way of the spilled result.	The cell in the way is emptied.

Table 3.2.13: Error values, example formulas, and ways to fix them.

Two more errors can be added to this list. `#NUM!` appears in a numerically invalid operation; for example `=SQRT(-4)` cannot take the square root of a negative number. `#NULL!` appears where the intersection of two ranges is expected but there is none, and it usually comes from typing a space instead of a colon.

An important property of error values is that they propagate. Every formula that uses a cell with an error gives an error too. A `#VALUE!` in B2 makes the column total that adds B2, and the report cell that uses that total, erroneous as well. That is why in a chain of errors the fix starts at the cell at the head of the chain.

3.2.8.3 Finding and fixing errors

When a cell with an error is selected, a small warning icon appears next to it. The icon's menu explains the type of error and offers an option for following the calculation step by step. The **Formula Auditing** group on the Formulas tab also helps find the source of the error: the **Trace Precedents** command shows with arrows which cells the selected formula uses. By following these arrows you reach the head of the chain.

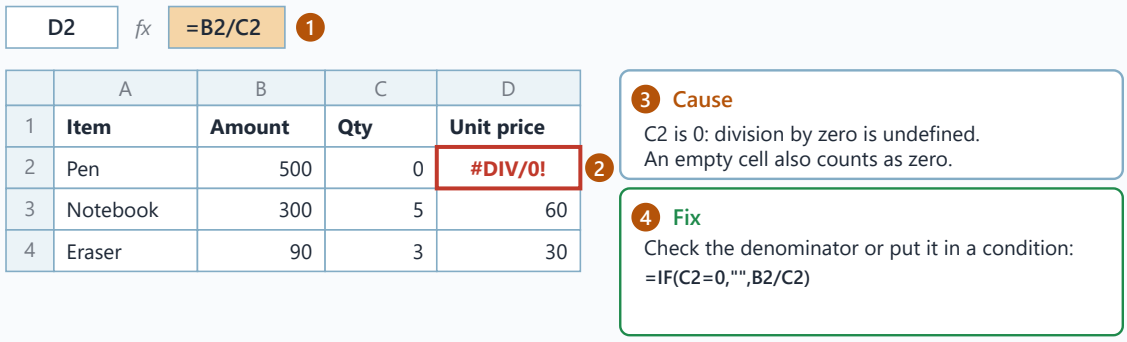


Figure 3.2.27: Division by zero in a unit-price calculation. In the formula `=B2/C2` in D2, C2 = 0, so #DIV/0! appears. The other rows are 300 / 5 = 60 and 90 / 3 = 30. Fix: `=IF(C2=0,"",B2/C2)`.

Example 3.2.17

Below the table in Figure 3.2.27, `=AVERAGE(D2:D4)` is typed into D5. What appears? How is the error removed?

1. **Propagation.** D2 holds #DIV/0!. AVERAGE cannot skip an error value in the range; **#DIV/0!** appears in D5 as well.
2. **Source.** When D5 is selected and Trace Precedents is used, the arrows point to D2:D4; used once more from D2, they point to B2 and C2. The problem is the 0 in C2.
3. **Fix.** D2 is changed to `=IF(C2=0,"",B2/C2)` and copied down. D2 now returns an empty text.
4. **New result.** AVERAGE ignores the text and averages only 60 and 30: (60 + 30) / 2 = **45**.

IF returning an empty text is a deliberate choice. Had it returned 0, AVERAGE would count that 0 as well and give a misleading result such as (0 + 60 + 30) / 3 = 30.

The most common cause of the #REF! error is deleting a column used by formulas. Excel cannot put another cell in place of the deleted cell and marks the reference inside the formula as #REF!. This error is fixed by undoing the deletion immediately with `Ctrl+Z`. If the values in a column are no longer needed, clearing its content instead of deleting the column preserves the references.

Common mistake

Mistaking a cell that shows ##### for an error. It is not an error value; because a number or date does not fit in the column, Excel fills it with marks instead of cutting it off. The value is stored correctly and formulas keep using it. When the column is widened, the value appears. The only exception is a negative date or time result; in that case ##### remains however wide the column is made.

3.2.9 Data-entry tools

Tools that speed up data entry and cut down repetitive work are the second leg of using Excel efficiently. This section covers AutoFill, commonly used shortcuts, Flash Fill, and the Excel table.

3.2.9.1 AutoFill

Definition 3.2.19

AutoFill is the extension of the value, formula, or pattern in the selected cells to adjacent cells when the fill handle is dragged. If Excel recognizes the pattern in the selection, it produces a **data series**; if it does not, it copies the value.

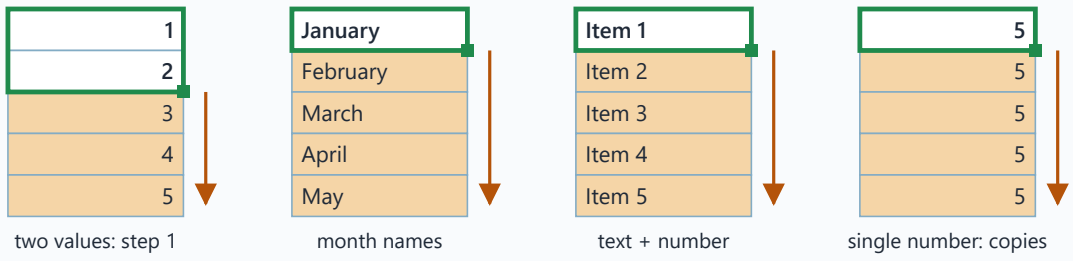


Figure 3.2.28: Series with the fill handle. When 1 and 2 are selected, the step is detected as 1 and 3, 4, 5 follow. “January” continues the month names and “Item 1” the text-and-number pattern. A 5 selected on its own is copied.

Excel works out the series from the first two values of the selection. If 2 and 5 are selected and dragged, the step is calculated as 5 − 2 = 3 and 8, 11, 14 follow. If 10 and 20 are selected, 30, 40, 50 follow. With dates each drag advances one day; day names such as Monday and month names such as January are also recognized. If a single number is selected and dragged, it is copied; if Ctrl is held while dragging, a series increasing by 1 is created.

When the drag ends, the **AutoFill Options** button appears at the bottom right. This button can be used to change the result: Copy Cells, Fill Series, Fill Formatting Only, or Fill Without Formatting. When a formula is dragged, relative references change according to the rule described in 3.2.5.2. In a long column, double-clicking the fill handle instead of dragging is also enough: the formula is copied automatically down to the last row where the column to its left is filled.

> **Common mistake**
>
> Not checking the first and last values of a filled series. For example, when a course code such as “CS-101” is dragged, Excel takes the trailing number for a pattern and produces “CS-102”, “CS-103”; yet the aim was to copy the code. Looking at the last cell after filling shows at once whether the pattern was understood correctly.

3.2.9.2 Shortcuts and Flash Fill

Action	Shortcut
Undo	`Ctrl+Z`
Format Cells	`Ctrl+1`
Create a table	`Ctrl+T`
Turn the filter on / off	`Ctrl+Shift+L`
Flash Fill	`Ctrl+E`
Find and Replace	`Ctrl+H`
Go to the end of the data block	`Ctrl+arrow key`

Table 3.2.14: Shortcuts frequently used in data work.

The shortcut in the last row saves a lot of time in long lists. `Ctrl+↓` jumps from the active cell to the last filled cell where the data ends. `Ctrl+Shift+↓` goes the same way while selecting; a column of thousands of rows is thus selected with a single keystroke.

Flash Fill learns the pattern in a column from an example. Let column A hold full names such as “Ayşe Yılmaz”, “Mehmet Demir”, and “Zeynep Kaya”. “Ayşe” is typed by hand in B2, then `Ctrl+E` is pressed in B3. Excel guesses the pattern as “the part up to the first space” and fills column B with Mehmet and Zeynep. The command can also be given from **Data → Data Tools → Flash Fill**.

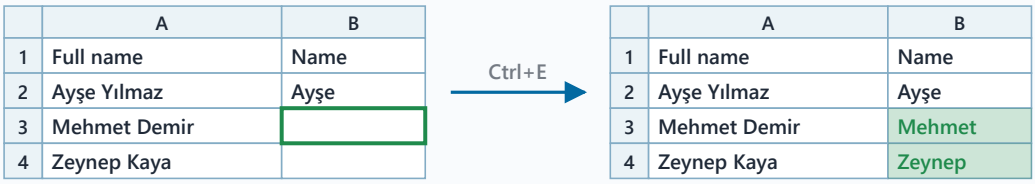


Figure 3.2.29: Flash Fill. "Ayşe" is typed in B2 as an example and Ctrl+E is pressed in B3; Excel guesses the pattern and fills in the names Mehmet and Zeynep by itself. The results are values, not formulas.

Flash Fill has two limits. First, the results are fixed values, not formulas; when a name in column A changes, column B is not updated. Second, Excel guesses the pattern, and the guess may not be right on every row. For a person with two given names such as "Ali Rıza Can", only "Ali" may come. The filled rows should therefore be reviewed one by one.

Find and Replace (`Ctrl+H`) replaces a text with another throughout the sheet. Before a bulk replacement, the scope of the search should be checked with the **Options** button: only the selected area, the whole sheet, or the whole workbook? If "Match entire cell contents" is not checked, a search for "Ali" also finds and replaces "Alican".

3.2.9.3 The Excel table

A data range with headers can be turned into a smart structure with Excel's **table** feature. A cell inside the range is selected and the **Insert → Table** command is given, or `Ctrl+T` is pressed. In the box that opens, the option stating that the first row is the header is checked and OK is clicked.

Definition 3.2.20

An **Excel table** is a range made of columns with headers that has been turned into a named, automatically expanding data structure. The columns of the table can be referred to in formulas by name instead of by address; this is called a **structured reference**.

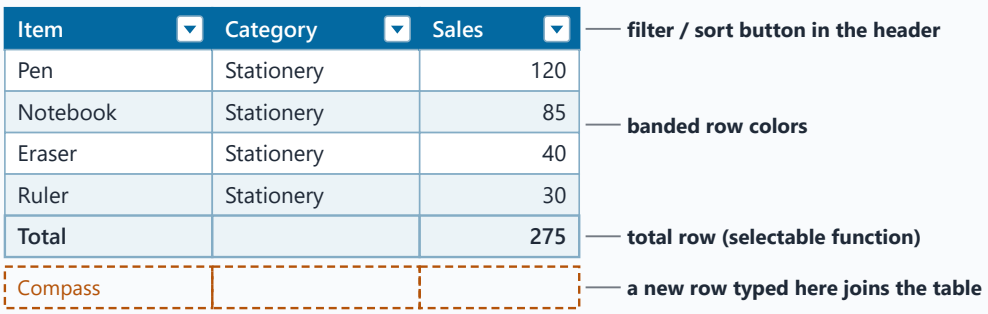


Figure 3.2.30: An Excel table. The headers have filter and sort buttons, there are banded row colors, and a total row shows the sum of sales (275). A new row (Compass) typed right below the table joins the table automatically.

The benefits of converting to a table are these:

- **Automatic expansion.** A new row or column typed right below or to the right of the table joins the table. Formulas, charts, and PivotTables that refer to the table cover the new data too.
- **Filter and sort.** A drop-down button is added to every header cell.
- **Structured reference.** If the table's name is Table1 and one of its columns is Sales, `=SUM(Table1[Sales])` adds the whole Sales column. Within a row, the value in the same row is referred to as `=[@Qty]*[@Price]`. When the formula is typed into one cell, it is copied automatically to the whole column.
- **Total row.** When the Total Row box on the **Table Design** tab is checked, a summary row is added below the table. For each column one of the functions such as sum, average, or count is chosen.
- **Style.** Banded rows and consistent colors make reading easier.

The sales values in Figure 3.2.30 are 120, 85, 40, and 30; the total row shows 120 + 85 + 40 + 30 = 275. As soon as a sales value is typed in the Compass row, the total row updates too. Had the same table been a plain range, the range of the total formula would have had to be extended by hand.

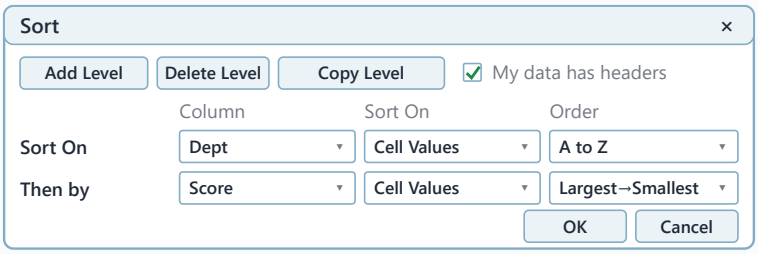
3.2.10 Sorting and filtering

The first step in drawing information out of a list is to arrange it according to the question at hand. Sorting changes the order of the records; filtering shows only certain records. Neither changes the data itself; but if sorting is done wrongly, it can corrupt the data permanently.

3.2.10.1 Sorting

> **Definition 3.2.21**
>
> **Sorting** is the rearrangement of the records in a list according to the values of one or several columns, keeping each row intact. Sorting by several columns is called a **multi-level sort**.

Sort direction is named according to the data type. Text is sorted from A to Z or Z to A, numbers from smallest to largest or largest to smallest, and dates from oldest to newest or newest to oldest. To sort by a single column, a cell in that column is selected and the A→Z or Z→A button on the Data tab is pressed. For a multi-level sort the **Data → Sort** dialog is opened.



Result

Dept	Score
Biology	88
Biology	71
Chemistry	92
Chemistry	64
Mathematics	80

Dept A→Z; then Score high→low

Figure 3.2.31: The Sort dialog. The first level is Dept, A to Z; the second level, added with **Add Level**, is Score, largest to smallest. On the right the result: Biology 88, 71; Chemistry 92, 64; Mathematics 80.

In the Sort box the **My data has headers** option must be checked. This option keeps the header row out of the sort. If the check mark is removed, the headers “Name, Dept, Score” are sorted like data and may end up mixed into the middle of the list. Excel also sorts numbers and numbers stored as text in the same column as separate groups; inconsistent data types spoil a sort.

With Turkish language settings, text is sorted according to the Turkish alphabet: Ç comes after C, Ğ after G, I after H, İ after I, Ö after O, Ş after S, and Ü after U. When the names Selin, Can, Çağla, Ali, İrem, and Burak are sorted A to Z, the result is Ali, Burak, Can, Çağla, İrem, Selin. A program that sorts letters by their code order on the computer throws Ç and İ to the end of the alphabet and gives the order Ali, Burak, Can, Selin, Çağla, İrem.

3.2.10.2 Keeping rows intact when sorting

In a list each row carries the whole information of one record. In sorting, these rows must move as a whole. Ali's department and score must stay on Ali's row after sorting too. Figure 3.2.32 shows the correct sorting of a five-student list by score.

	A	B	C	
1	Name	Dept	Score ↓	
2	Ece	Computer	91	(old row 3)
3	Can	Electrical	85	(old row 4)
4	Burak	Mechanical	78	(old row 6)
5	Ali	Electrical	72	(old row 2)
6	Deniz	Computer	64	(old row 5)

Sort: Score
Largest→Smallest
rows move as a whole

Figure 3.2.32: Sorting by Score, largest to smallest, with Data → Sort. The rows have been moved as a whole; each name stays with its own dept and score. The notes on the right show each row's old position.

Example 3.2.18

Ali (Electrical, 72), Ece (Computer, 91), Can (Electrical, 85), Deniz (Computer, 64), and Burak (Mechanical, 78) are sorted first by Dept from A to Z, and within the same dept by Score from largest to smallest. What is the result?

1. **First level.** The depts are in alphabetical order Computer < Electrical < Mechanical. The Computer students come first, then Electrical, and Mechanical comes last.
2. **Second level, only within the same dept.** Computer: Ece 91 > Deniz 64. Electrical: Can 85 > Ali 72. Mechanical has only one person: Burak 78.
3. **Result.** Ece, Deniz, Can, Ali, Burak.

The second level decides the order only among rows that are equal at the first level. Burak's score is higher than Deniz's and Ali's, but because Mechanical comes later in the alphabet, he stays at the end of the list.

The most dangerous error in sorting is selecting only one column and sorting. In this case Excel shows a **Sort Warning** and offers two options: **Expand the selection** and **Continue with the current selection**. If the second option is chosen, only that column is sorted; the other columns stay in place.

	A	B	C
1	Name ▾	Dept ▾	Score ↓
2	Ali	Electrical	91
3	Ece	Computer	85
4	Can	Electrical	78
5	Deniz	Computer	72
6	Burak	Mechanical	64

ERROR: rows scrambled
Only the Score column was selected, "Continue with current selection" was chosen

Figure 3.2.33: Wrong usage. Only the Score column was selected and "Continue with the current selection" was chosen. The scores have been arranged largest to smallest and the names stayed in place: Ali now shows 91 and Ece 85. The name–score pairing is broken.

Common mistake

Choosing "Continue with the current selection" in the sort warning. The resulting table looks tidy and there is no error message; but the records have been mixed up. If the file is saved and closed, the correct pairing can never be found again. In the warning, **Expand the selection** must always be chosen, and if the error is noticed it must be undone at once with Ctrl+Z.

3.2.10.3 Filtering

Definition 3.2.22

Filtering is showing only the rows in a list that meet certain criteria and temporarily hiding the others. A **filter** does not delete or change the data; it only narrows the view.

The filter is turned on by selecting a cell inside the list and using the **Data → Filter** command or `Ctrl+Shift+L`. A drop-down button appears on the right of every header cell. Clicking the button opens the list of values in that column. The rows of values whose check mark is removed are hidden.

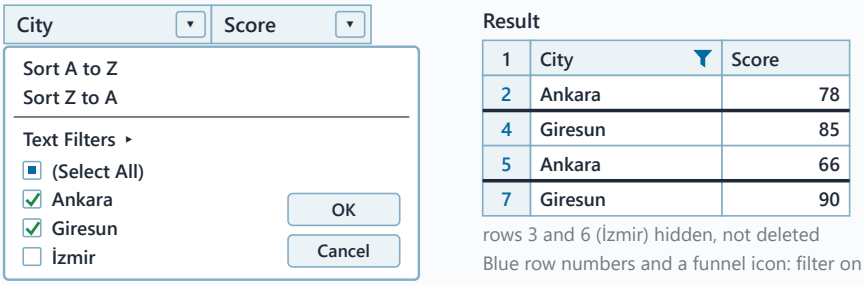


Figure 3.2.34: The filter menu of the City column. When İzmir's check mark is removed, rows 3 and 6 are hidden, not deleted. While the filter is on, the row numbers shown turn blue, and a funnel icon appears on the column header.

Besides the list of values, the filter menu also has condition filters. Depending on the column's data type, the menu shows **Text Filters** (Contains, Begins With, Equals), **Number Filters** (Greater Than, Less Than, Between, Top 10), or **Date Filters** (This Month, Last Year, Between). If filters are applied to several columns, the conditions narrow the list together: for a row to be visible it must meet the conditions in all the columns.

Example 3.2.19

In the six-student list (Ali Electrical 72, Ece Computer 91, Can Electrical 85, Deniz Computer 64, Burak Mechanical 78, Selin Electrical 58), first a Dept = Electrical filter and then a Score > 80 filter is applied. What is seen at each step?

1. **Start.** The filter has been turned on, but there is no condition: all 6 records are visible.
2. **Dept = Electrical.** Ali 72, Can 85, and Selin 58 are visible; 3 of 6 records. The row numbers become 1, 2, 4, 7 and proceed with gaps.
3. **Score > 80.** Number Filters → Greater Than → 80. Of the remaining three records only Can (85) meets the condition: 1 of 6 records.
4. **Clear. Data → Clear** removes all filters; all 6 records come back.

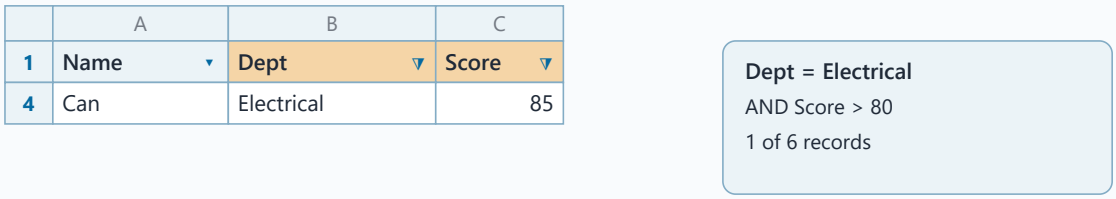


Figure 3.2.35: Two conditions together: Dept = Electrical and Score > 80. Only Can in row 4 is visible; both headers show a funnel icon.

In a filtered list, copying, deleting, and formatting affect only the visible rows. If the visible rows are selected and copied, the rows hidden by the filter are not copied; rows deleted while a filter is on are likewise only the visible ones. With manually hidden rows the situation is different: when such a range is copied, the hidden rows are copied too.

3.2.10.4 Filters and SUBTOTAL

What does a SUM formula below a filtered list give? The expected answer is the total of the visible rows. Yet SUM also adds the hidden rows. For a summary that respects the filter, the SUBTOTAL function is used.

Definition 3.2.23

SUBTOTAL is a function that calculates a summary value without counting the rows hidden by a filter: `=SUBTOTAL(function_num, range)`. The function number selects which calculation is done: 1 average, 2 count, 4 maximum, 5 minimum, 9 sum.

Let us take a new list of six records: Selin (Computer, 78), Can (Electrical, 85), Çağla (Computer, 91), Ali (Electrical, 64), İrem (Mechanical, 70), Burak (Computer, 55). The scores are in C2:C7. Three formulas are typed below the list:

```
C9:  =SUM(C2:C7)
C10: =SUBTOTAL(9,C2:C7)
C11: =SUBTOTAL(1,C2:C7)
```

	A	B	C
1	**Name**	**Dept**	**Score**
2	Selin	Computer	78
3	Can	Electrical	85
4	Çağla	Computer	91
5	Ali	Electrical	64
6	İrem	Mechanical	70
7	Burak	Computer	55

C9: =SUM(C2:C7) **443**
C10: =SUBTOTAL(9,C2:C7) **443**
C11: =SUBTOTAL(1,C2:C7) **73.8**

Figure 3.2.36: Before any filter is applied. All three formulas cover all the rows: SUM and SUBTOTAL(9) 443, SUBTOTAL(1) 73.8.

Step	Visible records	SUM	SUBTOTAL(9)	SUBTOTAL(1)
Start	all 6 records	443	443	443 / 6 ≈ 73.8
Sort: Dept A→Z, Score largest to smallest	Çağla 91, Selin 78, Burak 55, Can 85, Ali 64, İrem 70	443	443	73.8
Filter: Dept = Computer	Çağla 91, Selin 78, Burak 55	443	91 + 78 + 55 = 224	224 / 3 ≈ 74.7
Second filter: Score > 75	Çağla 91, Selin 78	443	91 + 78 = 169	169 / 2 = 84.5
Clear	all 6 records	443	443	73.8

Table 3.2.15: The values of the three formulas in the sort and filter steps.

Each row of the table verifies a rule. Sorting does not change the results, because the formulas point to the range C2:C7 and the values in the range have only changed places. A filter does not change SUM either; SUM also adds the hidden rows and always gives 443. SUBTOTAL, on the other hand, calculates only what is visible. When the filter is cleared all the rows come back and SUBTOTAL is 443 again.

	A	B	C	
1	Name ▾	Dept ▾	Score ▾	AND Score > 75
2	Çağla	Computer	91	
3	Selin	Computer	78	

C9: =SUM(C2:C7) 443
C10: =SUBTOTAL(9,C2:C7) 169
C11: =SUBTOTAL(1,C2:C7) 84.5

Figure 3.2.37: Two filters together: Dept = Computer and Score > 75. Only Çağla and Selin are visible. SUM is still 443; SUBTOTAL(9) gives 169 and SUBTOTAL(1) gives 84.5.

The function numbers of SUBTOTAL come in two groups, 1–11 and 101–111. The number 9 skips only the rows hidden by a filter. The number 109 also skips manually hidden rows. The total row of an Excel table uses the 109 version of SUBTOTAL behind the scenes; that is why, when the table is filtered, the total row also shows only what is visible.

3.2.11 Conditional formatting, PivotTables, and data validation

Making the important values in a list visible to the eye, summarizing the list by group, and preventing wrong data from being entered into it are three complementary jobs of data analysis. Excel offers conditional formatting, the PivotTable, and data validation for these jobs, respectively.

3.2.11.1 Conditional formatting

Definition 3.2.24

Conditional formatting is a rule that ties a cell's format to that cell's value. When the value changes, the format changes by itself.

The commands are collected in the **Home → Conditional Formatting** menu. The main rule types are:

- **Highlight Cells Rules.** Applies a format according to a certain threshold: Greater Than, Less Than, Between, Equal To, Text that Contains, Duplicate Values.
- **Top/Bottom Rules.** Marks the largest or smallest few values, or those above the average.
- **Data bars.** Draws a bar inside the cell proportional to the value.
- **Color scales.** Shows low and high values with a color gradient.

- **Icon sets.** Indicates a value's position in the range with icons such as arrows, circles, or flags.
- **Formula rule.** With **New Rule → Use a formula to determine which cells to format**, any formula that gives TRUE or FALSE can be written as the condition.

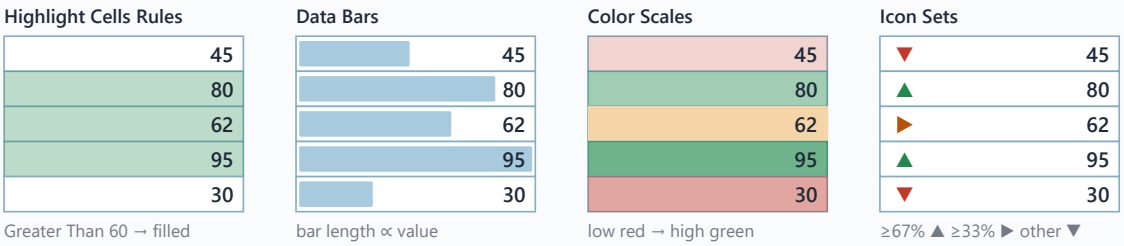


Figure 3.2.38: Four rule types applied to the same five values (45, 80, 62, 95, 30): highlighting those greater than 60, data bars, a color scale, and an icon set.

The formula rule is the most flexible type. For example, to color the *whole row* of a student list, A2:C7 is selected and the rule formula `=$C2<60` is typed. The formula is written relative to the top-left cell of the selection and applied as if copied to the other cells. `$C` locks the column, so each cell looks at the score in column C of its own row. Because the row number is free, each row tests its own score. This is another use of the mixed reference in 3.2.5.1.

Moderation is needed in conditional formatting. If five rules and ten colors are applied to the same range, the eye cannot pick out the important value. Color alone should not carry meaning; red and green are hard to tell apart for readers with color blindness. Writing a note beside a rule or using icons reduces this problem.

3.2.11.2 Thresholds of icon sets

Let the scores of seven students be in B2:B8: 72, 91, 85, 64, 78, 58, 45. Four different rules can be applied to these scores. If the cell rule “Less Than 60” is chosen, 58 and 45 get a red fill. In data bars the longest bar belongs to the largest value (91). In the color scale the lowest value (45) gets the lightest tone and the highest (91) the darkest. In the icon set, each value falls into one of three groups. But how are the groups decided?

The default thresholds of a three-icon set are 67% and 33%. These percentages are applied not to the values themselves but to the range between the smallest and the largest value:

> threshold = smallest + percentage × (largest − smallest)

Example 3.2.20

A three-icon set (▲ 67% and above, ● 33%–67%, ▼ below 33%) is applied to the scores 72, 91, 85, 64, 78, 58, 45. Let us find the thresholds and each score's icon.

1. **Range.** Smallest 45, largest 91; range 91 − 45 = 46.
2. **Upper threshold.** 45 + 0.67 × 46 = 45 + 30.82 = **75.82** ≈ 75.8.
3. **Lower threshold.** 45 + 0.33 × 46 = 45 + 15.18 = **60.18** ≈ 60.2.
4. **▲ cells.** 75.82 and above: 91, 85, 78.
5. **● cells.** Between 60.18 and 75.82: 72 and 64.
6. **▼ cells.** Below 60.18: 58 and 45.

The thresholds change with the data. If the lowest score drops from 45 to 30, the range widens to 91 − 30 = 61. The upper threshold falls to 30 + 0.67 × 61 ≈ 70.9 and the score 72 moves from ● to ▲; although 72 itself has not changed, its icon has. If fixed thresholds are wanted, in the **Manage Rules → Edit Rule** box the type is changed from "Percent" to "Number" and fixed values such as 75 and 60 are typed.

	A	B
1	Name	Score
2	Ali	● 72
3	Ece	▲ 91
4	Can	▲ 85
5	Deniz	● 64
6	Burak	▲ 78
7	Selin	▼ 58
8	Mert	▼ 45

Icon set
▲ ≥ 67% · ● ≥ 33% · ▼ other
thresholds: 75.8 and 60.2

Figure 3.2.39: Scores with an icon set applied. The upper third of the range (45–91) gets ▲, the middle ●, and the lower part ▼; the default 67% and 33% thresholds correspond to 75.8 and 60.2.

Common mistake

Mistaking the 67% in an icon set for "the best 67% of the scores". The threshold is computed not from the ranking of the values but from the distance between the smallest and the largest value. In the example three of the seven scores got ▲; that ratio has nothing to do with 67%. If a threshold by ranking is wanted, the rule type "Percentile" is chosen.

3.2.11.3 PivotTables

To summarize a list by group, a separate SUMIF or COUNTIF formula can be written for each group. As the number of groups grows, this way becomes tiring. The **PivotTable** does the same job without formulas, by dragging fields.

Definition 3.2.25

A **PivotTable** is an interactive table that places the columns of a list into row, column, filter, and value fields and produces summaries such as grouped totals, averages, and counts. It does not change the source data itself.

To create a PivotTable, a cell inside the list is selected and the **Insert → PivotTable** command is given. The source list must have headers and contain no empty rows. Excel opens an empty PivotTable on a new sheet and the **PivotTable Fields** pane on the right. The pane lists the headers of the source list as fields; below it are four boxes: Filters, Columns, Rows, and Values.

Example 3.2.21

The six-student list (Ali Electrical 72, Ece Computer 91, Can Electrical 85, Deniz Computer 64, Burak Mechanical 78, Selin Electrical 58) is summarized by department.

1. **Dept → Rows.** Each dept is listed once: Computer, Electrical, Mechanical, and at the bottom Grand Total.
2. **Score → Values.** The default summary of a numeric field is the sum. Computer 91 + 64 = 155, Electrical 72 + 85 + 58 = 215, Mechanical 78; Grand Total 155 + 215 + 78 = 448.
3. **Value Field Settings → Average.** Computer 155 / 2 = 77.5; Electrical 215 / 3 ≈ 71.7; Mechanical 78.0; Grand Total 448 / 6 ≈ 74.7.
4. **Name → Values.** The default summary of a text field is counting: Computer 2, Electrical 3, Mechanical 1; 6 people in total.

	A	B	C
1	**Name**	**Dept**	**Score**
2	Ali	Electrical	72
3	Ece	Computer	91
4	Can	Electrical	85
5	Deniz	Computer	64
6	Burak	Mechanical	78
7	Selin	Electrical	58

Fields

Rows:
Dept

Values:
Average of Score
Count of Name

Dept	Avg. Score	People
Computer	77.5	2
Electrical	71.7	3
Mechanical	78.0	1
Grand Total	**74.7**	**6**

Figure 3.2.40: The final state of the PivotTable. Rows hold Dept; values hold Average of Score and Count of Name. The Electrical average is 71.7 and its head count 3. In the Grand Total row the average is 74.7 and the head count 6.

The average in the Grand Total row needs attention. 74.7 is the average of the six scores: 448 / 6. The average of the three department averages, however, is (77.5 + 71.7 + 78) / 3 ≈ 75.7. That calculation counts the crowded Electrical group and the one-person Mechanical group with equal weight. The

PivotTable uses the right weighting itself.

A PivotTable and formulas give the same result: the Electrical total 215 can also be found with `=SUMIF(B2:B7,"Electrical",C2:C7)`. The difference is this: formulas update by themselves when the source data changes, but a PivotTable does not. When a new record is added to the source, the PivotTable is right-clicked and **Refresh** is chosen. If the source is defined as an Excel table, new rows are covered automatically on refresh.

3.2.11.4 Data validation

The quality of an analysis depends on the quality of the data entered. If 150 is typed in a grade column and “Electrcial” in a dept column, all later calculations are affected. Data validation makes it possible to catch wrong data at the moment it is entered.

Definition 3.2.26

Data validation restricts the values that can be entered into a cell with a rule. The rule can be a list, a number or date range, a text length, or a formula. The explanation shown when the cell is selected is called the **input message**, and the warning that opens on an entry that breaks the rule is called the **error alert**.

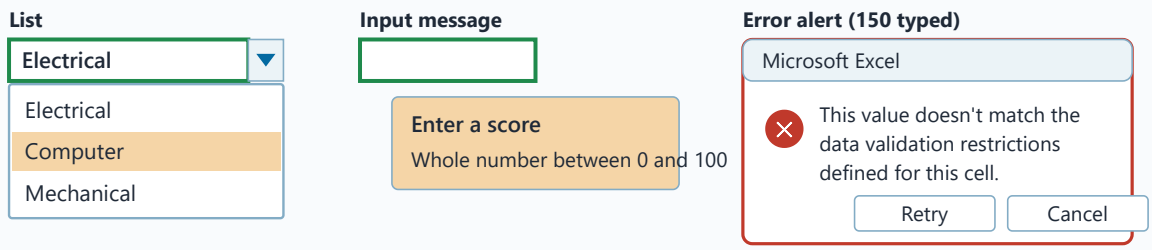


Figure 3.2.41: The three faces of data validation: a drop-down list in the dept cell, an input message “Whole number between 0 and 100” in the score cell, and the error alert that opens when 150 is typed.

Adding a drop-down list to a dept column takes six steps:

1. **Selecting the range.** The cells to be validated (for example B2:B4) are selected; **Data → Data Tools → Data Validation** is opened.
2. **The list rule.** On the **Settings** tab, **List** is chosen in the **Allow** box. In the **Source** box the options are typed separated by commas (`Electrical,Computer,Mechanical`) or a range holding the options is indicated (`=$E$2:$E$4`).
3. **Result.** When the cell is selected an arrow appears on its right; the user chooses only one of the values in the list.
4. **Error alert.** On the **Error Alert** tab the style, title, and message are written: Style Stop, title “Invalid group”, message “Choose a group from the list.”

5. **Invalid entry.** If “Banana”, which is not in the list, is typed, the Stop style rejects the entry; **Retry** returns the cell to editing.
6. **Choosing the style.** How strict the alert is depends on the style.

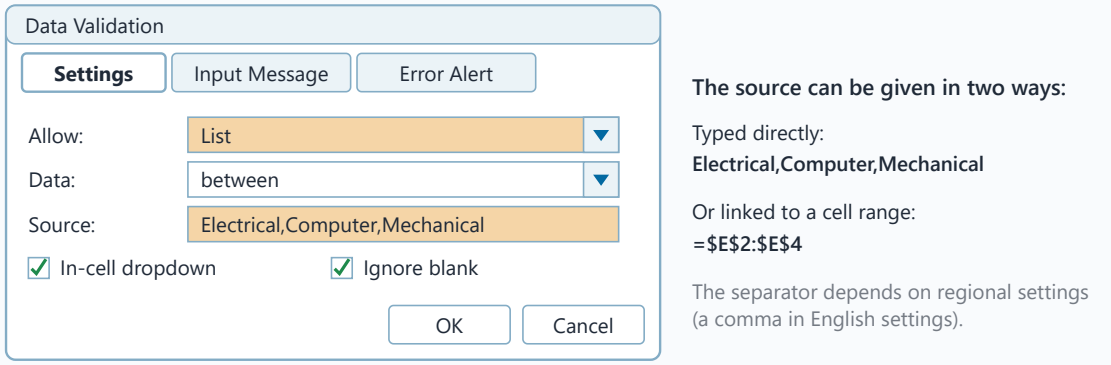


Figure 3.2.42: The Settings tab of the Data Validation box. Allow: List; Source: Electrical,Computer,Mechanical. The source can also be linked to a cell range (`=$E$2:$E$4`). The separator depends on regional settings; it is a comma in English settings.

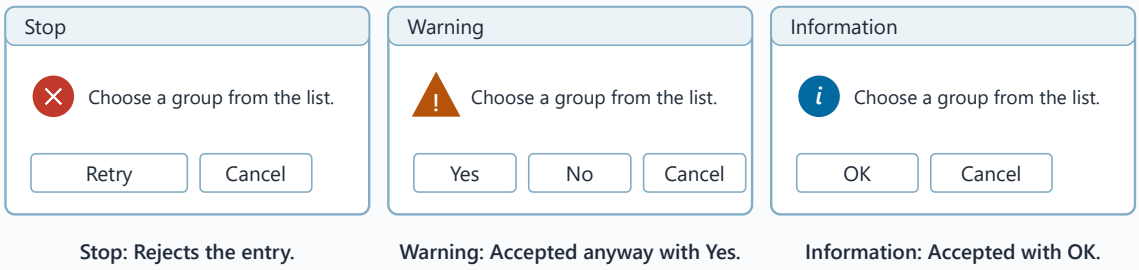


Figure 3.2.43: The three styles of the error alert. Stop rejects the entry. Warning accepts it anyway with Yes. Information only informs; the value is accepted with OK.

Linking the source to a cell range makes maintaining the list easier. When a new dept is added, only the list in column E is updated. The range is locked with $ signs; so when the rule is applied to several cells, each cell looks at the same list.

Data validation has a limit: the rule checks only entries typed by hand. Values copied from elsewhere and pasted can get past the rule. Invalid values entered earlier are not flagged automatically either. To find them, the **Data Validation → Circle Invalid Data** command is used.

3.2.12 Charts

A table gives exact numbers; a chart shows the trend, the comparison, and the outlier at a glance. But a chart of the wrong type or drawn carelessly can hide the data, or even leave a false impression. This section covers choosing the chart that suits the purpose of the data and making the chart clear.

3.2.12.1 Choosing a chart type

The chart type is chosen not by how the data looks but by the question the chart will answer. Table 3.2.16 matches the basic questions with the chart types that suit them.

Purpose	Suitable chart	Example question
Comparing categories	**Column or bar chart**	Which department has the higher average?
Showing change over time	**Line chart**	How have monthly sales changed?
Part-to-whole relationship	**Pie chart** with few categories, or a percent bar	What percentage of the budget went to materials?
Relationship of two numeric variables	**Scatter (XY) chart**	Does the grade rise as study time rises?
Shape of a variable's distribution	**Histogram**	In which ranges do the scores cluster?

Table 3.2.16: Choosing a chart by purpose.

A chart is inserted by selecting the range that holds the data and choosing a type from the Charts group of the **Insert** tab. The **Recommended Charts** button shows the types Excel suggests by looking at the data. In a column chart the categories go on the horizontal axis and the values on the vertical axis. A bar chart shows the same thing with horizontal bars and is more readable with long category names.

The pie chart is the most misused type. If the sum of the parts does not form a meaningful whole (for example the scores of five students), a pie chart is meaningless. When the number of categories goes beyond five or six, it becomes hard to compare the sizes of the slices by eye; in that case a sorted bar chart is better. The scatter chart and the line chart are also confused. In a line chart the horizontal axis is a category axis and the points are evenly spaced. In a scatter chart the horizontal axis is a true numeric axis. If the measurements are not evenly spaced, for example if they were taken at 1, 2, 5, and 10 seconds, a line chart distorts the time axis.

3.2.12.2 A clear and honest chart

In a good chart the reader sees at a glance what is shown. For this, the following elements of the chart are reviewed:

- **Title.** Says what the chart shows: not “Chart1” but “Monthly sales (thousand ₺)”.

- **Axes and units.** Axis titles state the meaning and unit of the values.
- **Source data.** It is checked that the right range and series are selected; a total row must not be accidentally drawn as a category.
- **Scale.** The axis limits are set so as not to make differences look bigger or smaller than they are.
- **Color.** Serves to separate series, but should not carry meaning alone.
- **Simplicity.** Three-dimensional effects, shadows, and needless decoration are removed; they make values harder to read.

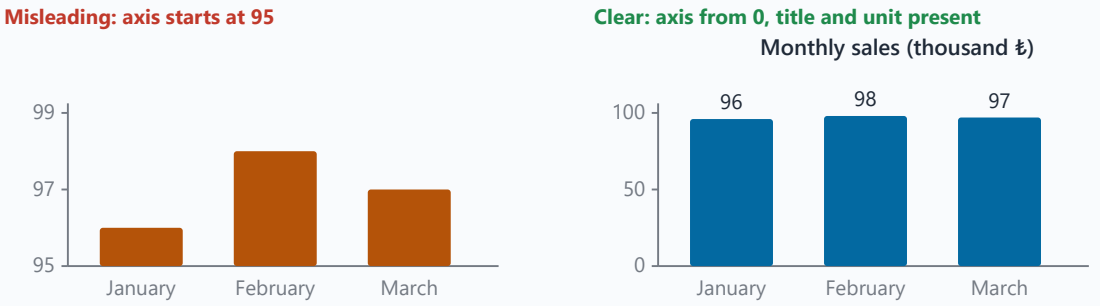


Figure 3.2.44: The same data in two charts. On the left the axis starts at 95 and small differences look very large. On the right the axis starts at 0 and there is a title and a unit: January 96, February 98, March 97 thousand ₺.

Example 3.2.22

January, February, and March sales are 96, 98, and 97 thousand ₺. If the vertical axis starts at 95, how many times as long does the February column look compared with the January column? What is the real difference as a percentage?

1. **Apparent heights.** Because the axis starts at 95, the column height is the part of the value above 95: January 96 − 95 = 1, February 98 − 95 = 3, March 97 − 95 = 2.
2. **Apparent ratio.** The February column looks **3 times** as long as the January column: 3 / 1.
3. **Real ratio.** 98 / 96 ≈ 1.021; February sales are only about **2.1%** more than January's.

In column and bar charts the axis must start at 0, because the reader compares the value with the length of the column. In a line chart, since the aim is to show the slope, the axis does not have to start at 0. Excel sometimes starts the axis from a value other than 0 by itself; this is fixed with the **Format Axis → Bounds → Minimum** setting.

3.2.13 Reporting, workbook management, and data layout

A workbook is often shared with others, printed, or used for years. This section covers preparing a workbook for printing, managing sheets, protecting it from change, and building a data layout suited to analysis from the start.

3.2.13.1 Printing and page layout

A table that looks fine on screen may be split into several pages when printed; the last column may land on a separate page. Before printing, the output is checked in the preview on the **File → Print** screen. The page-layout settings are collected on the **Page Layout** tab:

- **Print area.** Makes only the selected range print: Page Layout → Print Area → Set Print Area.
- **Orientation.** For wide tables, landscape is chosen.
- **Scaling.** If the Width box in the **Scale to Fit** group is set to "1 page", the table is shrunk to fit the page width.
- **Repeat titles.** In the box opened with the **Print Titles** button, row 1 is chosen as **Rows to repeat at top**; the header row is printed on every page of a multi-page printout.
- **Page breaks. View → Page Break Preview** shows with blue lines where the pages are split; the lines can be dragged to change the split.

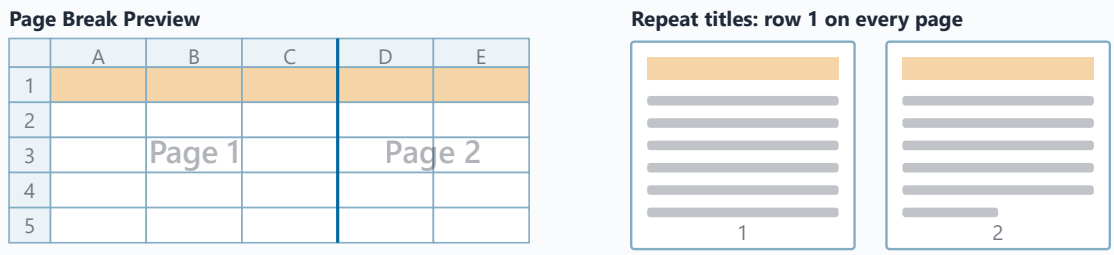


Figure 3.2.45: Page Break Preview. The blue dashed lines show where the pages are split. With the repeat-titles setting, row 1 is printed at the top of every page.

Scaling means shrinking, and shrinking has a limit. Let the printable width of a landscape A4 page be about 27.7 cm. When a table 35 cm wide is fitted to this width, the scale drops to 27.7 / 35 ≈ 79%. The text shrinks by the same ratio; 11-point text falls to about 8.7 points. If the scale falls below 60%, the printout becomes unreadable; in that case it is better to split the table or leave unneeded columns out of the print area.

3.2.13.2 Workbook management

Sheet operations are done from the menu that opens when a sheet tab is right-clicked: Insert, Delete, Rename, Move or Copy, Tab Color, Hide, and Unhide.

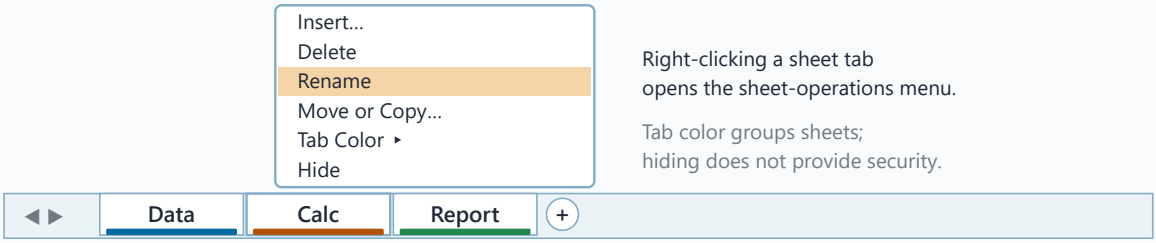


Figure 3.2.46: The sheet tabs (Data, Calc, Report) and the right-click menu. Tab color groups sheets; hiding does not provide security.

- **Renaming.** The tab is double-clicked and a short name that describes the content is typed. Sheet references in formulas update to the new name automatically.
- **Move or Copy.** Transfers the sheet to another position in the workbook or to another workbook. If "Create a copy" is checked, the sheet is copied, not moved.
- **Tab color.** Groups the data, calculation, and report sheets by color.
- **Hiding and unhiding.** Removes helper sheets from view. Anyone can bring a hidden sheet back with right-click → Unhide; so hiding is not a security measure.
- **Links.** Formulas that refer to another workbook depend on the location of that file. If the source file is moved or renamed, the link breaks. References of this kind are used carefully and sparingly.

3.2.13.3 Protection and collaboration

A sheet can be protected to prevent formulas from being deleted by accident. Sheet protection has two steps. The reason is that in Excel every cell is marked **Locked** by default; but this lock has no effect until protection is turned on.

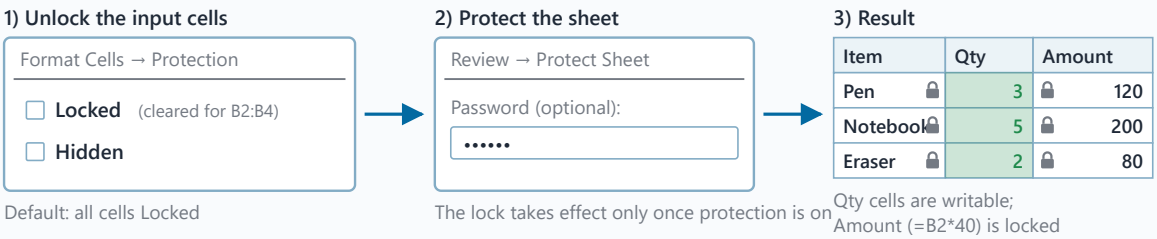


Figure 3.2.47: Sheet protection is two steps. First the Locked mark is cleared on the input cells the user will type into (B2:B4); then Review → Protect Sheet is turned on. As a result only the Qty cells are writable, and the Amount formulas stay locked.

1. **Unlocking the input cells.** The cells the user may change (for example the quantities) are selected. On the **Format Cells → Protection** tab, the **Locked** mark is cleared.
2. **Protecting the sheet.** The **Review → Protect Sheet** command is given; an optional password is typed. Now locked cells cannot be changed, and unlocked cells can be typed into.

In the example in Figure 3.2.47 the amounts are calculated with formulas such as `=B2*40`: 3 × 40 = 120, 5 × 40 = 200, 2 × 40 = 80. The user can change the quantities but cannot touch the formulas.

In Excel the levels of protection differ from one another:

- **Sheet protection** prevents cells and some operations from being changed. Its aim is to prevent accidental changes; it is not a strong security measure.
- **Workbook structure protection** (**Review → Protect Workbook**) prevents adding, deleting, and moving sheets and unhiding hidden sheets.
- **File encryption** (**File → Info → Protect Workbook → Encrypt with Password**) asks for a password to open the file. This is the real protection for files that carry confidential information. If the password is forgotten, the file cannot be recovered.

There are tools for collaboration too. **Comments** (Review → New Comment) are for asking and answering questions on a cell. If the file is stored in a cloud location such as OneDrive or SharePoint, several people can edit at the same time. **Version History** also makes it possible to go back to earlier states of the file.

3.2.13.4 Good data layout

Most of the tools described in these notes — sorting, filters, SUMIF, VLOOKUP, PivotTables — assume that the data is kept in a certain layout. If the layout is broken, the tools either do not work or silently give wrong results. A data sheet suited to analysis follows four rules:

- Each row represents a single record and each column a single variable.
- Headers are on a single row; they are short, unique, and meaningful. The unit is stated in the header: “Quantity (kg)”.
- Raw data has no empty rows, merged cells, or subtotals.
- Raw data, calculations, and report outputs can be kept on separate sheets.

✗ **Not suited to analysis**

	A	B	C
1	2024 Sales (merged)		
2	Item	Quantity	
3	Apple	12 kg	
4	(empty row)		
5	Subtotal	12 kg	

✓ **Tidy**

	A	B	C
1	Date	Item	Quantity (kg)
2	1/5/2024	Apple	12
3	1/5/2024	Pear	8
4	1/6/2024	Apple	15

totals in a separate cell / on a report sheet

Figure 3.2.48: On the left a layout unsuited to analysis: a merged header, an empty row, text with a unit (“12 kg”), and a subtotal. On the right tidy data: a single header row, numeric values, the unit in the header; totals in a separate cell or on a report sheet.

Example 3.2.23

Which Excel tools break on the table on the left in Figure 3.2.48? How are the same questions answered in the layout on the right?

1. **Merged header.** Sorting and filtering do not work because of the merged cell; converting to an Excel table does not recognize the headers either.
2. **Empty row.** Excel treats adjacent data as a whole. An empty row splits the list in two; Ctrl+T or a PivotTable takes only the upper part.
3. **The text "12 kg".** Because it is text it does not enter SUM, and in arithmetic it gives #VALUE!.
4. **Subtotal row.** When the column is summed, the subtotal is added once more and the result is doubled.
5. **In the tidy table.** The total for Apple is `=SUMIF(B2:B4,"Apple",C2:C4)` = 12 + 15 = 27 kg; all the quantities `=SUM(C2:C4)` = 12 + 8 + 15 = 35 kg. The totals sit not inside the data but in a separate cell.

These rules may look laborious at first. But a tidy data sheet saves struggling from scratch with every new question: the same list is sorted today, filtered tomorrow, and then becomes the source of a PivotTable. Another benefit of tidy data is that it transfers easily to other tools; a statistics program or a programming language can read this table as it is.

3.2.14 Summary

- A **workbook** consists of one or more **worksheets**. A cell address is written with the column letter and row number (C3); a **range** is indicated by its two corner cells (B4:D6 = 3 × 3 = 9 cells). A sheet has 1,048,576 rows and 16,384 columns.
- Excel left-aligns text and right-aligns numbers and dates; a left-aligned number has been stored as text and does not enter calculations. A **number format** changes only the display; calculations use the value in the formula bar.
- A **formula** begins with an equals sign and can contain constants, references, operators, and functions. **Operator precedence**: parentheses, negation, percent, exponent, multiplication–division, addition–subtraction, concatenation, comparison; left to right at the same level. In Excel `=-2^2` is 4.
- A **relative** reference (A1) shifts when copied, an **absolute** reference ($A$1) stays fixed, and in a **mixed** reference ($A1, A$1) only the part marked with $ stays fixed. F4 cycles through the types. In formulas that refer to a single rate cell, that reference must be absolute; otherwise Excel produces a wrong result without an error.

- SUM, AVERAGE, MAX, MIN, and COUNT ignore text and empty cells. **IF** returns one of two values according to a test; AND, OR, and NOT combine conditions; in a nested IF the conditions start from the high threshold. IFERROR hides the error but does not remove its cause.
- **COUNTIF** and **SUMIF** count and add by a criterion; for several conditions COUNTIFS and SUMIFS are used. **VLOOKUP** searches in the first column and returns a column to its right; in an exact match the last argument is 0. An approximate match is used in sorted scale tables. **XLOOKUP** can look in both directions; **FILTER** spills its result.
- Dates are **serial numbers**, times are fractions of a day; the difference of dates is found by subtraction. Each **error value** points to a cause: #DIV/0!, #NAME?, #VALUE!, #REF!, #N/A, #SPILL!. Errors propagate to the formulas that use them.
- **AutoFill** works out a series from the first two values; **Flash Fill** learns from an example but produces values. An **Excel table** expands by itself and allows structured references.
- In **sorting** the rows must move as a whole; in a multi-level sort the second criterion works only within equal groups. A **filter** hides rows, it does not delete them; conditions narrow together. SUM also adds hidden rows; **SUBTOTAL** calculates only what is visible.
- **Conditional formatting** ties format to value; the thresholds of icon sets are computed from the range between the smallest and the largest value. A **PivotTable** gives grouped summaries without formulas and must be refreshed when the source changes. **Data validation** prevents invalid entries from the start.
- The chart type is chosen by the question asked: columns for comparison, lines for time, a pie with few slices for part-to-whole, scatter for relationships, a histogram for the shape of a distribution. The axis of a column chart must start at 0.
- Before printing, the print area, scale, and repeated titles are set. Sheet protection prevents accidental changes; it does not provide secrecy, and for secrecy the file is encrypted. In a good data layout each row is a record and each column a variable; there are no empty rows, merged cells, or subtotals.

Calculation	Rule	Example
Number of cells in a range	(column difference + 1) × (row difference + 1)	C3:F10 → 4 × 8 = 32
Operator precedence	parentheses, sign, %, ^, * /, + –	`=2+3*(4-1)^2` → 29
Copying a formula	parts without $ shift by Δc, Δr	C3 → E7: `=A1+$B$2+B$1+$A2` → `=C5+$B$2+D$1+$A6`
Average	sum of the numbers ÷ count of numbers	12, 7, “absent”, 20, empty, 5 → 44 / 4 = 11
Approximate match	the largest threshold less than or equal to the lookup value	72 → threshold 70 → B
Date difference	difference of the serial numbers	2/1/2024 → 3/1/2024: 29 days
Hourly pay	duration (days) × 24 × hourly rate	1.875 × 24 × 150 = 6750 ₺
Icon threshold	smallest + percentage × (largest − smallest)	45 + 0.67 × 46 ≈ 75.8

Table 3.2.17: The main calculations used in the notes.

3.2.15 Key terms

workbook worksheet sheet tab active cell name box formula bar range data type logical value (TRUE/FALSE) freeze panes border fill wrap text number format formula constant operator operator precedence negation exponent concatenation (&) cell reference relative reference absolute reference mixed reference named range fill handle Show Formulas function argument logical test criteria exact match approximate match spill date serial number error value AutoFill data series Flash Fill table structured reference total row sort multi-level sort filtering SUBTOTAL conditional formatting data bars color scales icon sets PivotTable data validation drop-down list input message error alert column / bar chart scatter chart histogram print area sheet protection

3.2.16 Test Yourself

1. The formula `=A1+$B$2+B$1+$A2` in cell C3 is copied and pasted into D5. What is the formula in D5?

   Answer

   From C to D is 1 column right, from 3 to 5 is 2 rows down: Δc = 1, Δr = 2. `A1` → B3; `$B$2` does not change; `B$1` → C$1 (only the column shifts); `$A2` → $A4 (only the row shifts). Result: `=B3+$B$2+C$1+$A4`.

2. What do the formulas `=-3^2+10` and `=10-3^2` give in Excel? Why do they differ?

   Answer

   In the first, the leading minus is negation and is applied before the exponent: (−3)^2 = 9, 9 + 10 = 19. In the second the minus is subtraction and comes after the exponent: 3^2 = 9, 10 − 9 = 1. For the mathematical result $-3^2 + 10 = 1$, `=-(3^2)+10` must be written.

3. B2:B6 hold 40, “absent”, 70, empty, 90. What do `=AVERAGE(B2:B6)`, `=COUNT(B2:B6)`, and `=COUNTA(B2:B6)` give? What would the average be if 0 had been typed instead of “absent”?

   Answer

   Only 40, 70, and 90 are numbers: AVERAGE = 200 / 3 ≈ 66.7; COUNT = 3; COUNTA also counts the text: 4. If 0 is typed, the 0 enters the calculation as a number: 200 / 4 = 50.

4. In the six-student list in 3.2.7 (B2:B7 dept, C2:C7 score), what do `=COUNTIF(B2:B7,"<>Electrical")` and `=SUMIF(C2:C7,">=70")` give?

   Answer

   The ones other than Electrical are Ece, Deniz, and Burak: 3. Those with a score of 70 and above are 72, 91, 85, and 78; since no sum range is given, the scores themselves are added: 72 + 91 + 85 + 78 = 326.

5. With the scale 0 → F, 50 → D, 60 → C, 70 → B, 85 → A, what does the formula `=VLOOKUP(B2,$D$2:$E$6,2,TRUE)` return for the scores 59 and 84? What would happen if the last argument were 0?

   Answer

   59: the largest threshold less than or equal to it is 50 → D. 84: the largest suitable threshold is 70 (85 > 84) → B. In an exact match, neither is found in the table exactly, so #N/A is returned.

6. A project is run by working five days a week, from 08:30 to 17:30 each day. The daily duration is found with `=B2-A2` and five days are summed. If the total cell has the `h:mm` format, what appears? What is done for the correct display?

   Answer

   The daily duration is 17:30 − 08:30 = 9 hours; weekly 45 hours = 1.875 days. `h:mm` drops the whole day and shows 45 − 24 = 21 hours: 21:00. When the format code is set to `[h]:mm`, 45:00 appears.

7. In the list in 3.2.10.4 (Selin Computer 78, Can Electrical 85, Çağla Computer 91, Ali Electrical 64, İrem Mechanical 70, Burak Computer 55) a Dept = Electrical filter is applied. What do SUM, SUBTOTAL(9), and SUBTOTAL(1) give?

   Answer

   SUM also adds the hidden rows: 443. The visible rows are Can 85 and Ali 64: SUBTOTAL(9) = 149, SUBTOTAL(1) = 149 / 2 = 74.5.

8. The scores are 35, 50, 65, 80, 95, 100. The default three-icon set (67% and 33%) is applied. What

are the thresholds; which icon do 65 and 80 get?

Answer

The range is 100 − 35 = 65. Upper threshold 35 + 0.67 × 65 = 78.55; lower threshold 35 + 0.33 × 65 = 56.45. 80 ≥ 78.55 → ▲; 65 is between the two thresholds → ●. 35 and 50 get ▼, and 95 and 100 get ▲.

9. In a PivotTable the department averages come out as Computer 77.5 (2 people), Electrical 71.7 (3 people), Mechanical 78 (1 person). Why is the average in the Grand Total row 74.7 and not 75.7?

Answer

The overall average is the sum of the six scores divided by six: 448 / 6 ≈ 74.7. The simple average of the three averages, (77.5 + 71.7 + 78) / 3 ≈ 75.7, gives every department the same weight regardless of head count. The one-person Mechanical group would be overcounted and the three-person Electrical group undercounted.

10. In a name–score list a student selects only the Score column, sorts it largest to smallest, and says “Continue with the current selection” in the warning. What happens, and how is it fixed?

Answer

Only the scores change places and the names stay; each name is paired with someone else's score. Excel gives no error. It is undone at once with Ctrl+Z; in the sort warning “Expand the selection” is chosen so that the rows move as a whole.

3.2.17 Further reading

- Microsoft Support, Excel training pages (*support.microsoft.com*): short, up-to-date explanations with examples, in many languages, for formulas, functions, tables, PivotTables, and charts.
- M. Alexander and D. Kusleika, *Microsoft Excel 365 Bible* (Wiley, 2022): treats the functions touched on only briefly in these notes, dynamic-array formulas, PivotTables, and charts in reference-book detail.
- K. W. Broman and K. H. Woo, “Data Organization in Spreadsheets”, *The American Statistician* 72(1), 2018: a short, practical account of the rules for keeping data in a spreadsheet consistent, rectangular, and ready for analysis.
- C. Nussbaumer Knaflic, *Storytelling with Data* (Wiley, 2015): shows with examples how to choose the chart type, strip away needless elements, and direct the reader's attention to the right place.

3.2.18 References

1. Microsoft Support, *Basic tasks in Excel*.
2. Microsoft Support, *Overview of formulas in Excel*.
3. Microsoft Support, *Sort data in a range or table*.
4. M. Alexander, D. Kusleika, *Microsoft Excel 365 Bible*, Wiley, 2022.
5. K. W. Broman, K. H. Woo, "Data Organization in Spreadsheets", *The American Statistician*, 72(1), 2–10, 2018.
6. C. Nussbaumer Knaflic, *Storytelling with Data: A Data Visualization Guide for Business Professionals*, Wiley, 2015.

PART I · CHAPTER 3.3

Microsoft PowerPoint

A project proposal, a lab result, or a graduation project is often presented in a talk of a few minutes. On the screen behind that talk there is usually a PowerPoint deck. These notes cover all the stages from planning a talk to sharing the deck: the workspace and views, planning by time and the outline, file formats, slide layouts and sections, themes and the Slide Master, visual hierarchy, pictures, shapes, tables, and charts, alignment, transitions and animation, speaker notes and Presenter View, rehearsing timings, accessibility, inspection, and printing. Every topic is supported by figures and step-by-step tables. Numbers such as animation times, picture size, compression, alignment coordinates, and page count are explained with small calculations.

LEARNING OUTCOMES

Planning
Determine the purpose, audience, and main message of a talk; calculate the number of slides from the speaking time; build the content in Outline View.

Slide management
Use layouts, themes, the Slide Master, sections, hidden slides, and zoom appropriately; explain the precedence between manual formatting and the master-slide rule.

Visual communication
Place text, pictures, shapes, tables, and charts legibly; calculate picture size and compression savings; align objects and distribute them at equal spacing.

Delivering the talk
Distinguish a transition from an animation and calculate the start and end times of animations; use speaker notes, Presenter View, rehearsed timings, and show shortcuts; run accessibility and technical checks before sharing the deck.

3.3.1 PowerPoint and the workspace

Microsoft PowerPoint is the presentation program in the Microsoft 365 suite. Word produces a text document and Excel a spreadsheet; PowerPoint produces a file made of pages that appear on screen one after another. Each of these pages is called a slide. These notes are written for the English interface of Microsoft 365 PowerPoint on Windows. In the web, Mac, and mobile versions of PowerPoint the location of menus may differ a little, but the concepts are the same.

3.3.1.1 What is presentation software for?

Two closely related terms are used with different meanings. The file itself is called a *deck* (or presentation file), and the act of delivering it in front of an audience is called a *talk*. For example, one says “the deck has 20 slides” but “the talk lasted 10 minutes”. This distinction is kept throughout the notes.

Definition 3.3.1

A **slide** is a page that fills the screen at one time; it can carry text, pictures, shapes, tables, charts, audio, and video. A **presentation file** (a **deck**) is a file made of slides that will be shown in a particular order. A **talk** is the delivery a speaker gives accompanied by these slides.

PowerPoint is used most of all to support a live talk. But its uses are not limited to that (Table 3.3.1).

Use	Role of the deck	Example
Lectures and talks	Supports the speaker's delivery visually	A lecture, a seminar talk
Project presentation	Structures purpose, method, findings, and conclusion in a particular order	A graduation project defense
Reporting	Summarizes data and findings with charts	A monthly energy-consumption report
Training material	Organizes steps, examples, and exercises	Training on how to use a lab instrument
Information screen (kiosk)	Advances by itself with nobody touching it and returns to the start	The announcement screen at the faculty entrance
Recording and video	Produces an output that can be watched together with narration	A lecture video for distance education

Table 3.3.1: The main uses of PowerPoint.

What these uses have in common is this: a slide does not replace what is being said; it supports it. The slides of a good deck hold the key points, numbers, and visuals. The detail is in the speaker's words, in the speaker notes, or in a separate document.

3.3.1.2 The workspace

When PowerPoint is opened and a new deck is created, five main regions share the screen (Figure 3.3.1). At the top is the ribbon, on the left the slide pane, in the middle the slide area, below it the notes pane, and at the bottom the status bar.

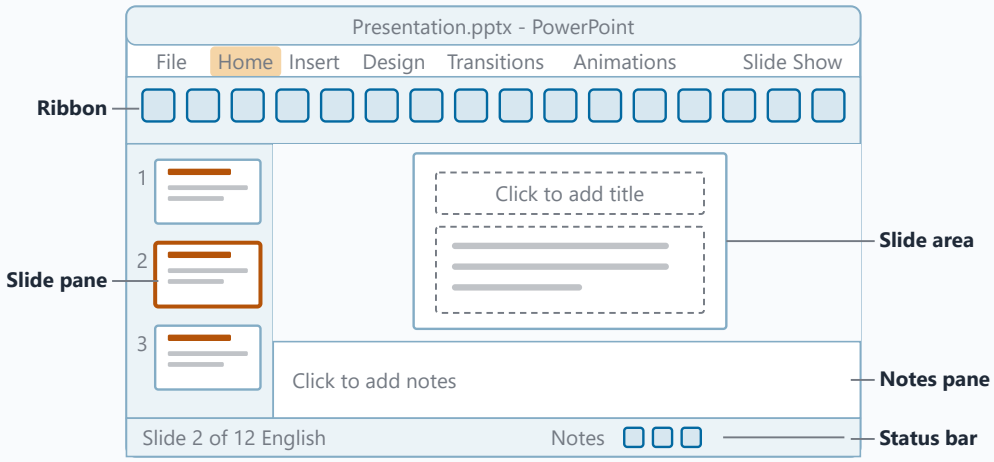


Figure 3.3.1: The PowerPoint window. At the top the ribbon with tabs, on the left the slide pane showing thumbnails of the slides, in the middle the slide being edited, below it the notes pane, and at the bottom the status bar ("Slide 2 of 12").

Definition 3.3.2

The **ribbon** is the tool area that shows commands divided into tabs such as File, Home, Insert, Design, Transitions, Animations, Slide Show, Review, and View. In each tab the commands are grouped by function. A **contextual tab** is a tab that appears only while a particular object is selected; for example, when a picture is selected the Picture Format tab appears, and when a table is selected the Table Design tab.

The roles of the regions are summarized in Table 3.3.2. Two of them are especially important. The slide pane is used to manage the deck as a whole, and the notes pane is for the speaker.

Region	What it shows	What it is used for
Ribbon	Tabs and command groups	Insert, format, transition, animation, and show commands
Slide pane	Numbered thumbnails of the slides	Selecting, reordering, duplicating, deleting, and hiding slides
Slide area	A large view of the selected slide	Typing and editing the content
Notes pane	The speaker note for the selected slide	Writing explanations that do not fit on the slide, the timing, and reminders

Region	What it shows	What it is used for
Status bar	Slide number, language, view buttons, zoom	Seeing the position, switching quickly between views

Table 3.3.2: The regions of the PowerPoint workspace.

In the dashed boxes in the slide area, texts such as “Click to add title” appear. These boxes are called placeholders; they do not appear in the show and only indicate what is to be written where. The type and position of each box are determined by the slide's layout.

Common mistake

Looking on the Home tab for the commands needed to edit a picture. Commands such as Crop, Compress, and Remove Background are on the Picture Format contextual tab, and this tab is visible only while the picture is selected. If the command being looked for is not on the ribbon, first check whether the relevant object is selected.

3.3.1.3 Presentation views

The same deck can be shown in different ways for different jobs. PowerPoint calls these views. Views are chosen from the View tab or with the buttons in the status bar. Changing the view does not change the content of the file; it changes only how it is shown on screen.

View	What is on screen	When it is used
Normal	The slide pane, the slide, and the notes	When editing slide content; the default view
Outline View	Headings and bullets as text	When building the content as text, before design
Slide Sorter	All the slides in a grid	To see the flow as a whole, to reorder, to divide into sections
Notes Page	The slide on top, the notes below, in page layout	When preparing speaker notes in a form that will be printed
Reading View	The deck plays inside the window	Trying out the show without going full screen
Slide Master	The master slide and the layouts tied to it	Managing the common design that affects the whole deck
Slide Show	Slides full screen, one at a time	When presenting to the audience

Table 3.3.3: PowerPoint views and their purposes.

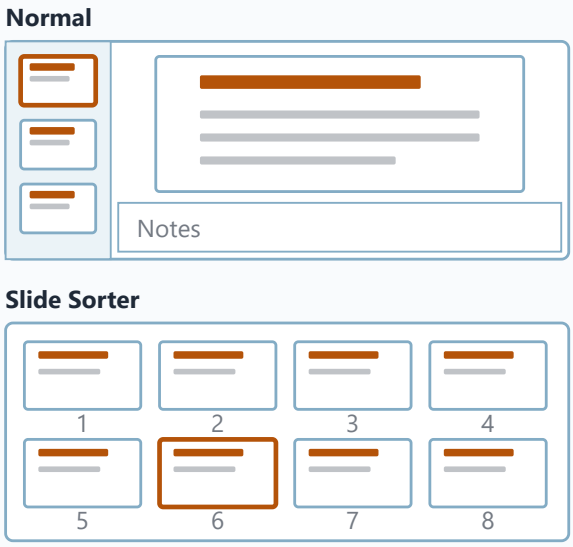


Figure 3.3.2: Two views of the same deck. In Normal view there are thumbnails on the left, the slide being edited on the right, and the notes below. In Slide Sorter all eight slides appear in a grid.

Figure 3.3.2 shows the difference between two views. Normal view suits working on a single slide. Slide Sorter shows the deck as a whole: how many slides each section holds, which slides resemble one another, and where the flow breaks are easily noticed in this view.

Example 3.3.1

A student is working on a 35-slide graduation deck. Which view is the most suitable for each of the following jobs?

1. The advisor said, “The method slides should come before the findings.” The position of several slides will change. All the slides need to be seen at the same time and moved by dragging: **Slide Sorter**.
2. There are no slides yet; only the headings and bullets are known. The content will be entered quickly, as text: **Outline View**.
3. The university logo will go in the bottom-right corner of all the slides: the common design changes instead of slide by slide, **Slide Master** view.
4. The speaker notes will be printed on paper and held in the hand at rehearsal: to see the layout of the notes on the page, **Notes Page**.

None of the four jobs is impossible in Normal view; but each is done several times faster in the suitable view.

3.3.2 Planning the talk

Starting to prepare a deck by opening PowerPoint is a common habit. Yet the most important part of a good talk is the plan made before the first slide is drawn. The plan answers three questions: What is this talk for? Who is it for? If the audience will remember a single sentence as they leave the room, which sentence is it?

3.3.2.1 Purpose, audience, and main message

The elements of the plan, in order, are: purpose, audience, main message, evidence and examples, flow, and closing. The purpose says why the talk is given: to inform, to persuade, to get a decision made, or to teach. The audience determines how the same subject will be presented. For example, a solar-panel project is presented to faculty jury members with technical detail and to high-school students with examples from daily life.

Definition 3.3.3

The **main message** is the one-sentence conclusion that all the parts of the talk support. “Sensor-controlled lighting reduces the laboratory's electricity use by about a third” is a main message; “Lighting systems” is merely a topic heading.

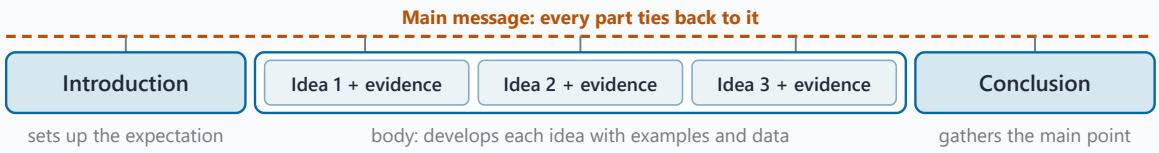


Figure 3.3.3: The flow of a talk. The introduction sets up the expectation, in the body each idea is developed with evidence, and the conclusion gathers the main point. The main message ties all the parts together.

The flow usually has three parts (Figure 3.3.3). The introduction builds an expectation in the audience by saying why the subject matters. The body develops, in order, a few ideas that support the main message; beside each idea is a piece of evidence, a number, or an example. The conclusion states the main message again, plainly. Three practical rules come out of this structure:

- Every slide has a single, clear purpose. If a slide holds two separate ideas, two slides are usually needed.
- Slides are not a transcript of the talk. If every sentence the speaker will say is written on the slide, the audience stops listening and starts reading.
- The closing gives no new information; it gathers the main message. A “Thank you” slide alone is not a closing.

Common mistake

Mistaking the title of the talk for the main message. A title such as “Energy Use in the Laboratory” does not tell the audience what to think. A main message contains a judgment and can be supported by a number: “Half of the laboratory's night-time consumption comes from devices left on.”

3.3.2.2 A slide plan based on time

Talks often have to fit a particular time. That is why the number of slides is calculated from the time before the content. There is a rule of thumb found through experience: a content slide carries on average 1–1.5 minutes of speech. Slides that are not talked over, such as a title slide or a sources slide, are not counted in this calculation.

Example 3.3.2

A 15-minute session is given for a project talk. Part of the session will be set aside for questions and answers. A slide plan is to be drawn up.

1. The question-and-answer time is written into the plan first: a 15-minute session = 10 minutes of talk + 5 minutes of questions.
2. The 10 minutes are shared among the parts: introduction and problem 2 min, method 3 min, findings 4 min, conclusion 1 min. The total is 2 + 3 + 4 + 1 = 10 minutes.
3. Each part is given a number of slides that fits its time (Table 3.3.4). The total of the content slides is 2 + 3 + 3 + 1 = 9.
4. It is compared with the rule of thumb: 10 ÷ 9 ≈ 1.1 minutes/slide. This value is within the range 1–1.5; the plan is realistic.
5. The title and sources slides are added. The file has 11 slides, but 9 enter the time calculation.

Part	Time	Slides	Content
Introduction and problem	2 min	2	The topic and why it matters
Method	3 min	3	One slide per step
Findings	4 min	3	One finding and one chart per slide
Conclusion	1 min	1	Repetition of the main message
Total	10 min	9	+ title and sources slides

Table 3.3.4: A time-based slide plan for a 10-minute talk.

In the findings part, three slides fall to four minutes; because the charts are interpreted in the slides of this part, the time per slide is a little longer. The rule of thumb is not a strict rule but a warning threshold. Its real use is that it catches unrealistic plans at once. For example, if 30 slides have been prepared for 10 minutes, each slide gets 600 ÷ 30 = 20 seconds. In that time a chart can only be shown, not explained.

Some details may be needed if a question comes up: a long results table or the intermediate steps of a calculation. These are added to the end of the deck as hidden slides. Hidden slides are skipped in the normal show and opened when needed.

Common mistake

Estimating the time only by looking at the number of slides and not rehearsing at all. The rule of thumb is only a starting point. The real time is measured only by a spoken rehearsal; PowerPoint's Rehearse Timings tool records this measurement slide by slide.

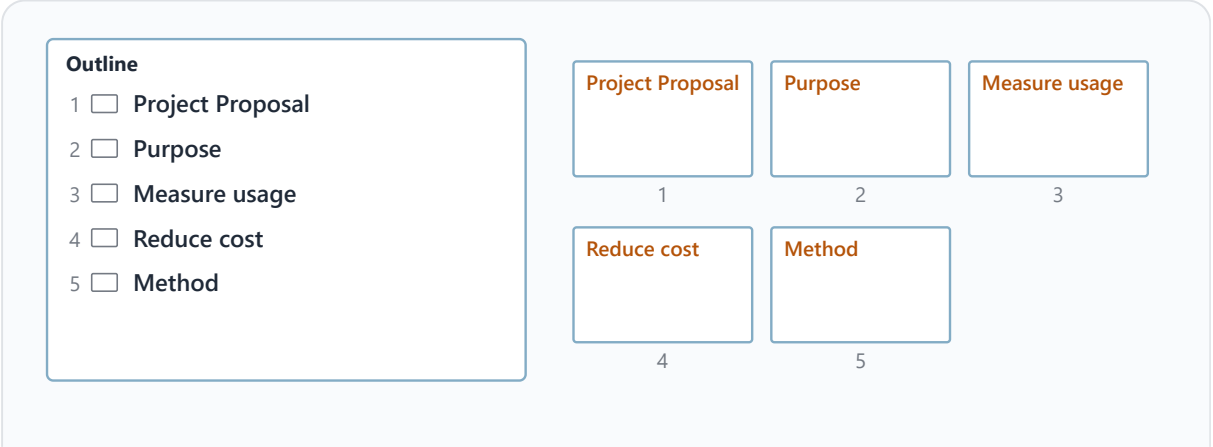
3.3.2.3 Outline View

The fastest way to pour the plan into slides is View → Outline View. In this view the deck looks like an indented list of text. Every line has a level. A line at the top level is the title of a slide; a line one level lower is a bullet of that slide.

Definition 3.3.4

The **outline** is the display of a deck's titles and bullets as text together with their levels. In the outline, Enter opens a new line at the same level; Tab lowers the level of the line by one (turns a title into a bullet), and Shift + Tab raises it by one (turns a bullet into a title).

The real power of the outline is this: as the levels change, the number of slides changes by itself. Figure 3.3.4 shows how a five-line outline takes shape in three steps. The operations are followed in order in Table 3.3.5.



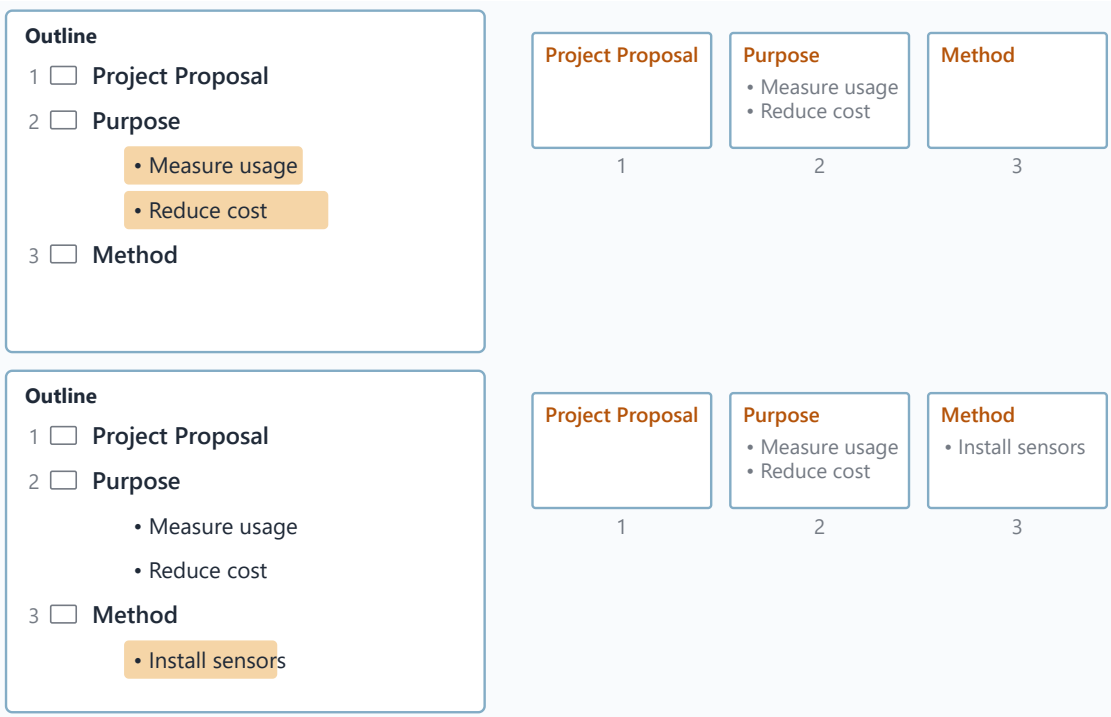


Figure 3.3.4: Levels in Outline View. At the top each of the five lines is a slide title: five slides. In the middle Tab was pressed on lines 3 and 4; these lines became bullets of the "Purpose" slide and the slide count fell to three. At the bottom a bullet was added after "Method" with Enter + Tab.

Step	Action	Outline	Slide count
1	Five lines are typed, with Enter after each	Project Proposal / Purpose / Measure usage / Reduce cost / Method	5
2	Tab on the "Measure usage" line	"Measure usage" becomes a bullet of Purpose	4
3	Tab on the "Reduce cost" line	"Reduce cost" also becomes a bullet of Purpose	3
4	Enter at the end of the "Method" line, then Tab	The new line becomes a bullet of Method; "Install sensors" is typed	3

Table 3.3.5: Level operations in Outline View and the slide count.

The fourth step needs care. When Enter is pressed at the end of a title line, the new line opens at the same level as the title; that is, a new, empty slide is created at that moment. When Tab is pressed, this empty slide turns into a bullet of the previous slide and the slide count falls to three again. In the outline, the only thing that determines the slide count is the number of top-level lines.

The outline applies the principle of content first, design second. While titles and bullets are being typed, font, color, and pictures are not thought about. After the flow has settled, a layout and a theme are applied to the slides. That way the effort spent on design does not go to slides that will be deleted later.

Common mistake

Forgetting to press Tab when meaning to type a bullet in the outline. Every new line opened with Enter is at title level; five lines typed thinking they were bullets create five separate slides. If the slide pane shows an unexpected number of empty-looking slides, the first place to look is the levels in the outline.

3.3.3 File and slide management

When the plan is ready, it is time for the file and the slides. This section covers the format the deck will be saved in, the templates from which slides will be created, and how to keep slides in order in a long deck.

3.3.3.1 New deck and file formats

A new deck is started empty or from a template with File → New. The file format chosen when the deck is saved determines how the file will open on another computer. The same content behaves very differently in different formats (Figure 3.3.5).

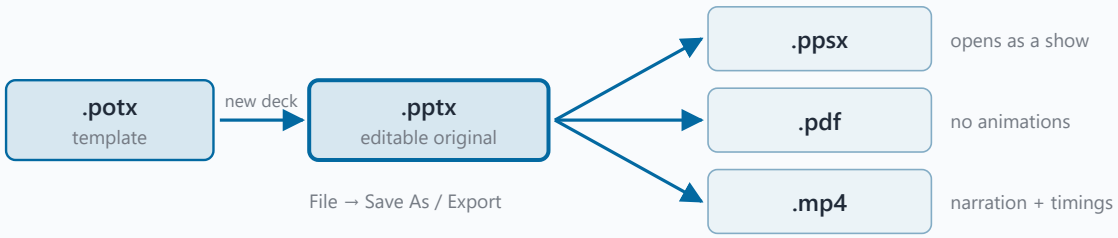


Figure 3.3.5: The relationships among deck file formats. A new .pptx deck is created from a .potx template. The editable original is .pptx; from it, Save As or Export produces a .ppsx show file, a .pdf, and an .mp4 video.

Extension	What kind of file?	On double-click	Animation
.pptx	Standard, editable deck	PowerPoint opens in Normal view	Yes
.ppsx	Show file	The slide show starts directly	Yes
.potx	Template	A new, untitled deck opens from the template	Yes
.pdf	Fixed-layout document	The pages appear in a PDF reader	None; the final state of the slide
.mp4	Video	Plays in a video player	Plays according to the timings

Table 3.3.6: The main file formats of PowerPoint.

Definition 3.3.5

A **template** is a file, used as the starting point for new decks, that carries a theme, layouts, and sample content. When a template is opened, what is edited is not the template itself but an untitled copy derived from it; the template is thus not damaged.

The difference between Save and Save As is also important. Save overwrites the open file. Save As creates a new file; the original stays unchanged and PowerPoint now works on the new file. Saving a deck as .ppsx or .pdf does not delete or convert the original .pptx file.

Example 3.3.3

A student has prepared a graduation deck (deck.pptx). The deck will go to three places. Which format is suitable for each?

1. The advisor will read the deck and correct the slides: it must be editable, so **.pptx** is sent.
2. A printed copy will be given to the jury members: the page layout must stay the same on every printer, so **.pdf**. A PDF has no animations; every slide is printed in its final state.
3. It will loop as a promotion on the screen at the faculty entrance: the show must start directly when opened and nobody should see the editing view, so **.ppsx**. If the computer does not have PowerPoint, or can only play video, **.mp4** is chosen.

All three outputs are produced from the same .pptx file with Save As or Export. If a correction is needed later, only the .pptx is corrected and the other formats are produced again.

Common mistake

Keeping the deck only as a PDF. Editing slides from a PDF is laborious, and animations and speaker notes are lost. Whatever the sharing format, the editable .pptx file is always kept separately.

3.3.3.2 Slide layout and placeholders

A slide can be started from a blank page, but PowerPoint offers ready-made patterns instead. These patterns are called layouts. A layout determines which boxes a slide will have and where these boxes will sit.

Definition 3.3.6

A **placeholder** is a dashed-frame box on a slide reserved for a particular kind of content: title, text, content (text, table, chart, or picture), picture, and so on. A **slide layout** is the pattern that defines the type, number, and position of the placeholders.

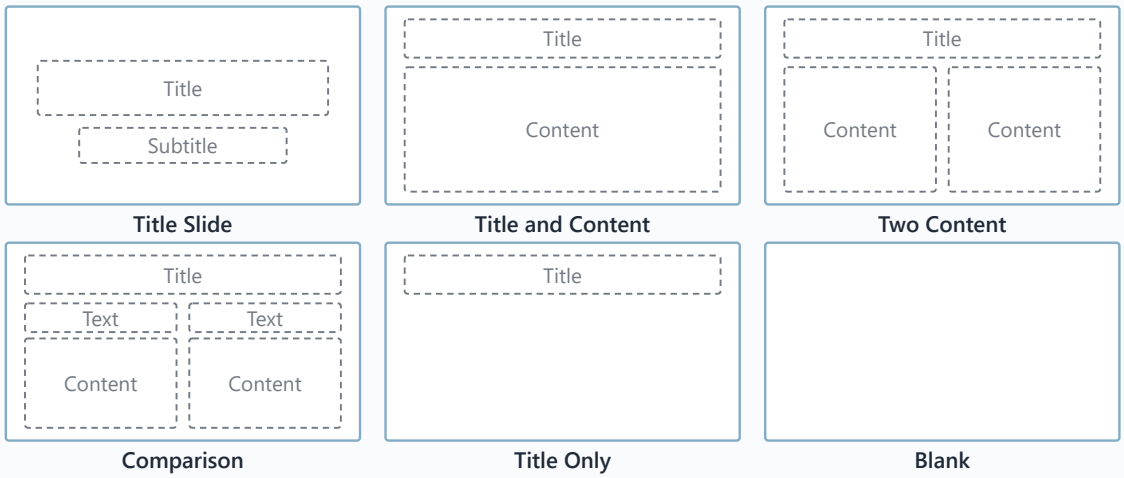


Figure 3.3.6: Six common slide layouts: Title Slide, Title and Content, Two Content, Comparison, Title Only, and Blank. The dashed boxes are placeholders.

Figure 3.3.6 shows the six most frequently used layouts. The Title Slide is for opening the deck. Title and Content is the ordinary slide carrying a title and a bullet list or a single chart. Two Content puts two things side by side; Comparison adds a heading to each of them. Title Only leaves an empty area beneath to fill freely. The Blank layout has no placeholders at all.

The layout of a slide can also be changed later with Home → Layout. In that case the text on the slide does not disappear; it moves into the suitable placeholders of the new layout. For example, if a slide in the Title and Content layout is turned into Two Content, the bullets move to the left box and the right box stays empty.

Using placeholders has three benefits. First, titles sit in the same place and in the same format on every slide; the audience's eye does not search for the title on each slide. Second, the text in a placeholder shows in Outline View; text in a hand-drawn text box does not. Third, screen readers read the slide title from the placeholder. A slide whose title is typed in a free text box looks “untitled” to a screen reader.

Common mistake

Opening every slide with the Blank layout and adding the title and bullets one by one with text boxes. Although the slides look tidy at first glance, the position of the titles drifts by a few pixels from slide to slide. In the outline these slides appear without titles, and when the theme

changes the text boxes do not fit the new design.

3.3.3.3 Adding, duplicating, moving, and hiding slides

The basic slide operations are done in the slide pane, by right-clicking the thumbnails, or from the Home tab (Table 3.3.7).

Action	How	Result
New slide	Home → New Slide or Ctrl + M	Added after the selected slide, with the chosen layout
Duplicate	Right-click → Duplicate Slide or Ctrl + D	A copy of the selected slide is added right after it
Delete	Select the thumbnail and press Delete	The slide is removed from the deck and the numbers are reassigned
Move	The thumbnail is dragged in the slide pane or in Slide Sorter	The order changes and the numbers are reassigned
Section	Home → Section → Add Section	Slides are divided into named groups
Hide	Right-click → Hide Slide	The slide stays in the file and is skipped in the normal show

Table 3.3.7: Basic slide operations.

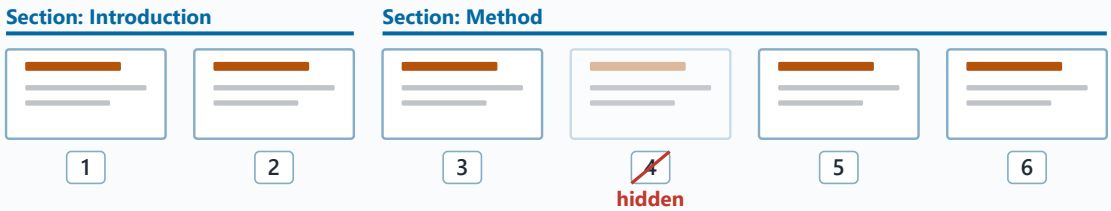


Figure 3.3.7: Two sections (Introduction and Method) and a hidden slide 4 in Slide Sorter. The thumbnail of the hidden slide appears faded and its number struck through.

Slide numbers are not attached to the slide; a number shows the slide's current position. When a slide is deleted or moved, all the numbers after it shift. A hidden slide, however, keeps its number but is skipped in the show (Figure 3.3.7).

Example 3.3.4

A deck has slides P, Q, R, S in order. The following operations are performed in the slide pane. After each step, what is the slide order, and which slides appear in a show started with F5?

Step	Action	Order in the file	Slide count
0	Start	P Q R S	4
1	S is selected, Ctrl + D	P Q R S S′	5

Step	Action	Order in the file	Slide count
2	Q is selected, Delete	P R S S′	4
3	P is selected, Ctrl + M (new slide Y)	P Y R S S′	5
4	S′ is dragged and dropped before P	S′ P Y R S	5
5	R is selected, Hide Slide	S′ P Y R(hidden) S	5

Table 3.3.8: A trace of operations in the slide pane.

In the show, four slides appear in the order S′, P, Y, S; R is skipped. In the file, however, there are five slides. After step 5 the number of R remains 4 and that of S is 5. R's content has not been deleted; the hidden state of the slide can be removed again with Hide Slide.

Common mistake

Treating a hidden slide as deleted. A hidden slide is inside the file: anyone who receives the file can open it and see it. Content that must not be shared is removed by deleting, not by hiding.

3.3.3.4 Sections

In a deck of forty or fifty slides the slide pane becomes a long strip and navigating gets hard. Sections solve this problem. A section is a group of consecutive slides that has a name. Sections are only for editing and navigation; they are not visible in the show.

Definition 3.3.7

A **section** is a group of consecutive slides shown with a name in the slide pane and in Slide Sorter. A section starts at the selected slide and runs until the start of the next section. A section can be moved, collapsed, renamed, and deleted as a whole.

The greatest benefit of sections is moving in bulk. A group of slides left in the wrong place is put where it belongs without dragging the slides one by one. Figure 3.3.8 and Table 3.3.9 give an example step by step. In a seven-slide deck the method slides (Survey, Sample) have by mistake ended up after the findings.

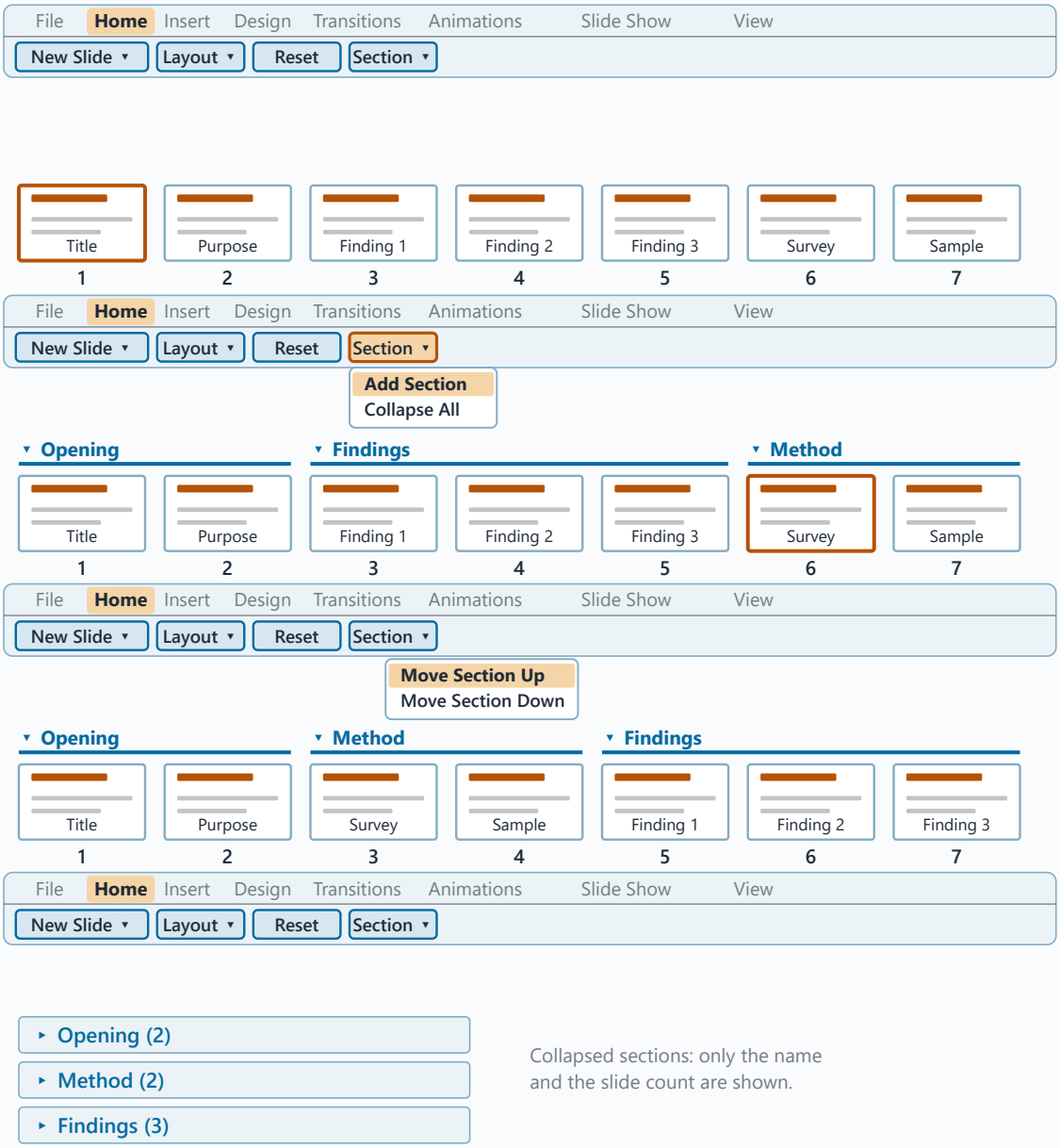


Figure 3.3.8: Adding and reordering sections. In the first frame there are seven slides without sections. In the second frame the Opening, Findings, and Method sections have been opened. In the third frame Move Section Up has put the Method section ahead of the findings. In the fourth frame Collapse All shows only the section names and slide counts.

Step	Action	Sections (slide numbers)	Slide order
1	Start	No sections	Title, Purpose, Finding 1, Finding 2, Finding 3, Survey, Sample
2	Slide 1 selected, Add Section, name “Opening”	Opening (1–7)	Unchanged
3	Slide 3 selected, Add Section, name “Findings”	Opening (1–2), Findings (3–7)	Unchanged

Step	Action	Sections (slide numbers)	Slide order
4	Slide 6 selected, Add Section, name “Method”	Opening (1–2), Findings (3–5), Method (6–7)	Unchanged
5	On the Method section, right-click → Move Section Up	Opening (1–2), Method (3–4), Findings (5–7)	Title, Purpose, Survey, Sample, Finding 1, Finding 2, Finding 3
6	Right-click → Collapse All	Opening (2), Method (2), Findings (3)	Unchanged

Table 3.3.9: Adding and moving sections.

Between the second and fourth steps the slide order never changes; adding a section only groups. The real change is in the fifth step. The Method section moves up one place together with the two slides inside it, and the numbers are reassigned: the number of Survey falls from 6 to 3, and that of Finding 1 rises from 3 to 5. To do the same job without sections, the two slides would have to be dragged separately, dropping each in the right place every time.

Common mistake

Thinking that section names will show in the show as subheadings. A section name is visible only on the editing screen. If you want the audience to see the change of section, a separate title slide is put at the start of the section.

3.3.3.5 Zoom and hyperlinks

A show normally runs linearly: 1, 2, 3, … Sometimes this order is not enough. In the question-and-answer period you may need to return quickly to a detail, or to choose one of the sections according to the audience's interest. PowerPoint has two tools for this: zoom and the hyperlink.

Definition 3.3.8

Summary Zoom is a contents slide made of clickable thumbnails of the chosen sections. Clicking a thumbnail goes to that section; when the section ends, the show returns to the summary slide. **Section Zoom** and **Slide Zoom** jump to a single section or slide. All are added with Insert → Zoom.

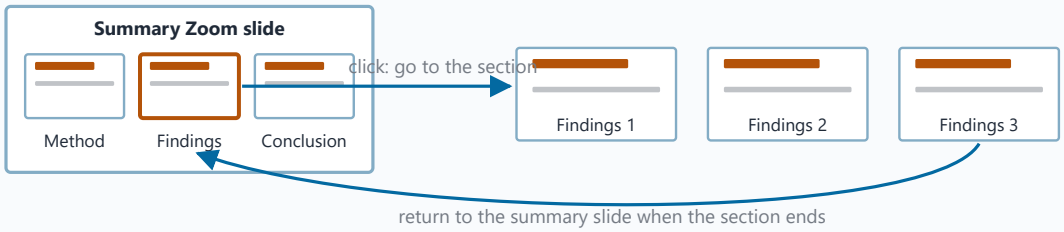


Figure 3.3.9: A Summary Zoom slide. Clicking the “Findings” thumbnail shows the three slides of the Findings section; when the section ends, the show returns to the summary slide.

A **hyperlink** ties an object or text to another destination: a slide in the same deck, another file, or a web address. A hyperlink is added with Insert → Link. Action buttons (Insert → Shapes → Action Buttons) are shapes with ready-made targets such as “next”, “previous”, “first slide”, and “home”. In a kiosk show that does not advance by clicking, hyperlinks and action buttons are the only tools that let the viewer navigate.

Zoom is available in current PowerPoint versions. When the deck is exported to PDF, the zoom thumbnails remain static pictures and the movement is not seen. A nonlinear flow built with zoom therefore becomes linear by itself in a PDF.

Common mistake

Relying on a web hyperlink during a talk and not preparing an offline backup. If the hall has no internet connection, the link goes nowhere. A screenshot of the page the link would open is put on a backup slide.

3.3.4 Theme, Slide Master, and design

The secret of a professional-looking deck is not decoration but consistency. All the titles sit in the same place, in the same font and the same color. The same colors mean the same thing on every slide. PowerPoint builds this consistency with two tools: the theme and the Slide Master.

3.3.4.1 Theme and Slide Master

Definition 3.3.9

A **theme** is the design set that defines a deck's color palette, title and body fonts, effects, and background together; it is chosen from the Design tab. The **Slide Master** is the topmost slide that carries the rules common to all layouts: the position of placeholders, the font, color, logo, footer, and so on.

The way to understand the Slide Master is to see rules flowing downward in a hierarchy (Figure 3.3.10). At the top is the slide master. Beneath it are the layouts tied to it (Title Slide, Title and Content, Two Content, …). At the bottom are the real slides that use those layouts. A rule on the slide master passes to the layouts, and a rule on the layouts passes to the slides.

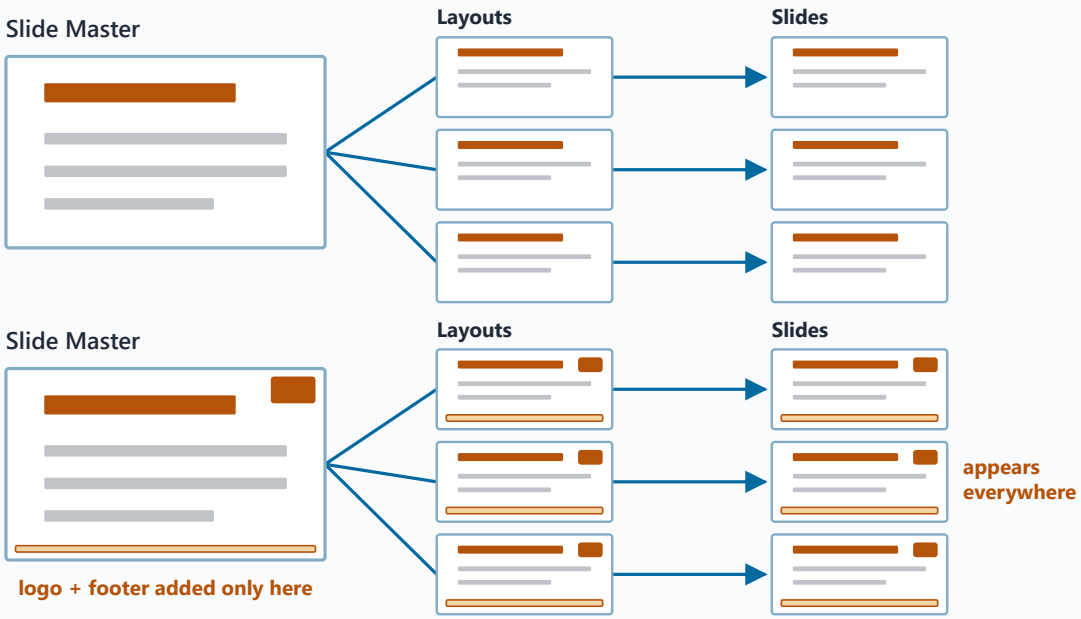


Figure 3.3.10: The Slide Master, layouts, and slides. At the top is the flow of rules from the slide master to the layouts and from the layouts to the slides. At the bottom a logo and a footer have been added only to the slide master and have appeared in the same place on all layouts and slides.

The Slide Master is entered with View → Slide Master. In this view, in the left pane, the slide master appears at the top with a large thumbnail and the smaller layout thumbnails beneath it. Items that must be on every slide, such as a logo, an institution name, or a slide number, are added here once. They also appear by themselves on slides added later.

The theme and the Slide Master work together. The theme chooses color and font; the Slide Master determines where and how these choices are applied on the slide. When the theme is changed, the slide master and all the layouts are redrawn according to the new theme.

Common mistake

Adding the logo to the first slide and duplicating that slide as many times as needed. Every logo created this way is a separate object. When the logo changes, it has to be changed on all the slides one by one; on slides added with a new layout, the logo does not appear at all. The place for an item that will be on every slide is the slide master.

3.3.4.2 Changes in the Slide Master and manual formatting

The rules of the slide master do not always win. Formatting given directly, by hand, to an object on a slide takes precedence over the rule coming from the slide master. If this precedence is not known, it is not understood why a change made in the Slide Master does not show on some slides.

Definition 3.3.10

Manual formatting is formatting given directly to an object on a single slide with commands on the Home tab such as font, color, and size. Manual formatting locks only the property that was changed; the other properties continue to come from the slide master. Home → Reset returns the placeholders of the selected slide to the layout's position, size, and formatting and deletes the manual formatting.

Table 3.3.10 follows an example step by step. In a three-slide deck the title of Slide 2 has earlier been made red by hand. The title color will be changed in the Slide Master, and a new layout with a picture area will also be defined.

Example 3.3.5

Step	Command	What happens	Title of slide 1 / 2 / 3
1	View → Slide Master	The Slide Master tab opens	theme color / red (manual) / theme color
2	Navy color on the master's title placeholder	The titles of all layouts turn navy	(not visible yet)
3	Insert Layout, Insert Placeholder → Picture	A picture area is drawn on the new layout, which has a title	(not visible yet)
4	Rename: “Title and Picture”	The layout will be listed under this name	(not visible yet)
5	Close Master View	Return to Normal view	navy / red / navy
6	Slide 2 selected, Home → Reset	The red given by hand is removed	navy / navy / navy
7	New Slide → Title and Picture	Slide 4 comes with the picture area	Slide 4 is navy too

Table 3.3.10: A change in the Slide Master and the precedence of manual formatting.

The result in the fifth step is the precedence rule itself. The titles of slides 1 and 3 had no manual color; their color comes from the slide master and becomes navy. The title color of slide 2 had been given by hand; the change in the slide master does not override this property, and it stays red. In the sixth step Reset removes this lock. The seventh step shows that the new layout also takes the navy rule from the slide master.

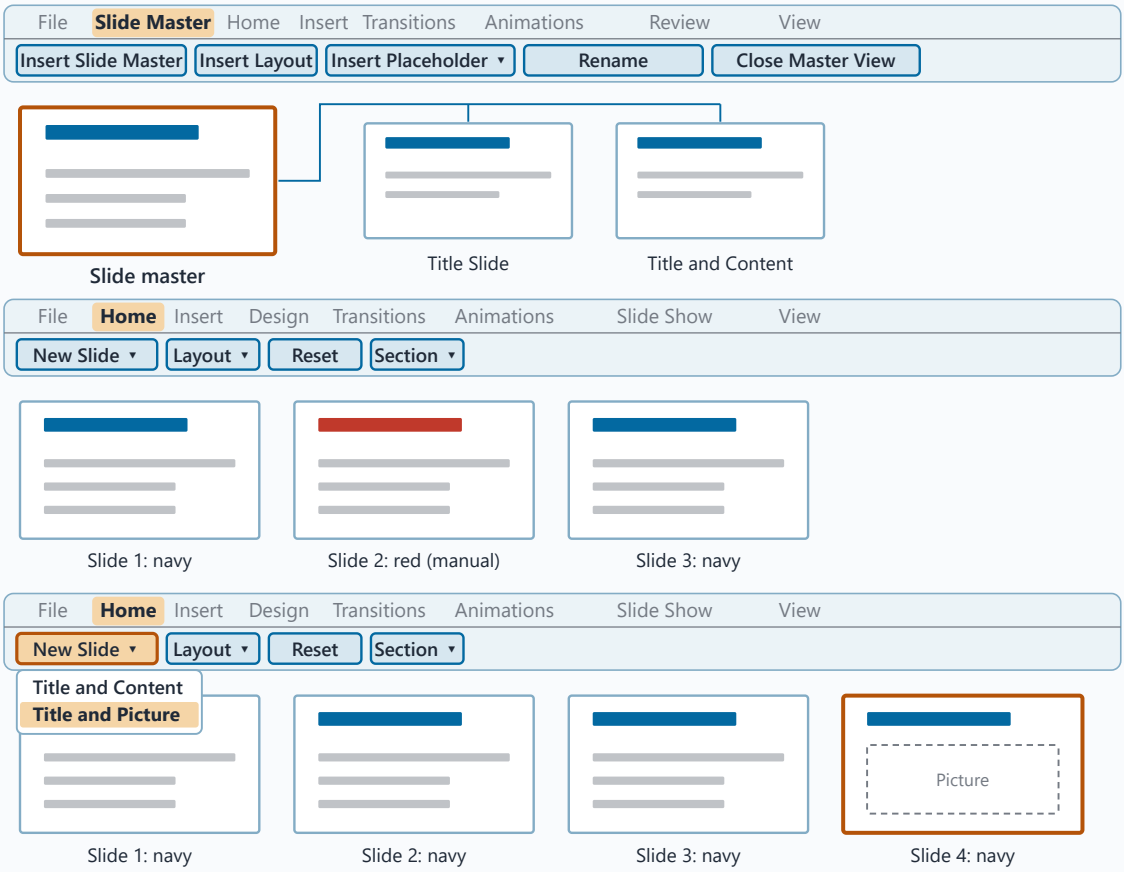


Figure 3.3.11: The Slide Master step by step. At the top the master's title has been made navy and this has carried through to all the layouts. In the middle, after Close Master View, Slide 2's manually given color is kept. At the bottom, after Reset, all three titles are navy, and Slide 4 has been added with the new "Title and Picture" layout.

Manual formatting locks only the property that was changed. Suppose a title was given only a different font by hand and its color was not touched. If both the font and the color are changed on the slide master, on this title the font stays as given by hand, but the color goes to the new slide-master color.

Common mistake

Repeating a change made in the Slide Master by hand on the slides that did not pick it up. This does not remove the lock; it adds a new one. The right fix is to select the slides that did not pick up the change and return them to the layout with Home → Reset.

3.3.4.3 Copying formats and bulk replacement

For cases that the theme and Slide Master do not cover, PowerPoint also has tools that copy formats or replace them in bulk (Table 3.3.11).

Tool	Tab	What it does
Format Painter	Home	Copies the font, color, and border format of the selected object to another object; if double-clicked it is applied to several objects, and released with Esc.
Animation Painter	Animations	Copies an object's animation effect and timing to another object.
Apply To All	Transitions	Gives the selected transition and its duration to all slides.
Reset	Home	Returns the placeholders' position, size, and formatting to the layout.
Replace Fonts	Home → Replace	Replaces a font used in the deck with another font everywhere.

Table 3.3.11: Tools for copying formats and bulk replacement.

The thinking behind these tools is the same. Manual formatting sits separately on each object, and when a change is needed each object is corrected one by one. The theme, the Slide Master, and the bulk tools, on the other hand, are changed once and the change reflects everywhere. For example, changing the font of the titles in a 40-slide deck is 40 separate operations with manual formatting and a single operation in the Slide Master.

Common mistake

Trying to apply the Format Painter to several objects with a single click. A single click applies it only to the next object and the tool turns off. For several objects the button is double-clicked; when the job is done, Esc is pressed.

3.3.4.4 Visual hierarchy

When the audience first looks at a slide, the eye goes somewhere. This first look lasts a few seconds. The designer's job is to make the eye go first to the most important information.

Definition 3.3.11

Visual hierarchy is showing the order of importance of the elements on a slide through size, weight, color, position, and space. The most important element is the largest or most prominent; detail is written smaller and source information faintest.

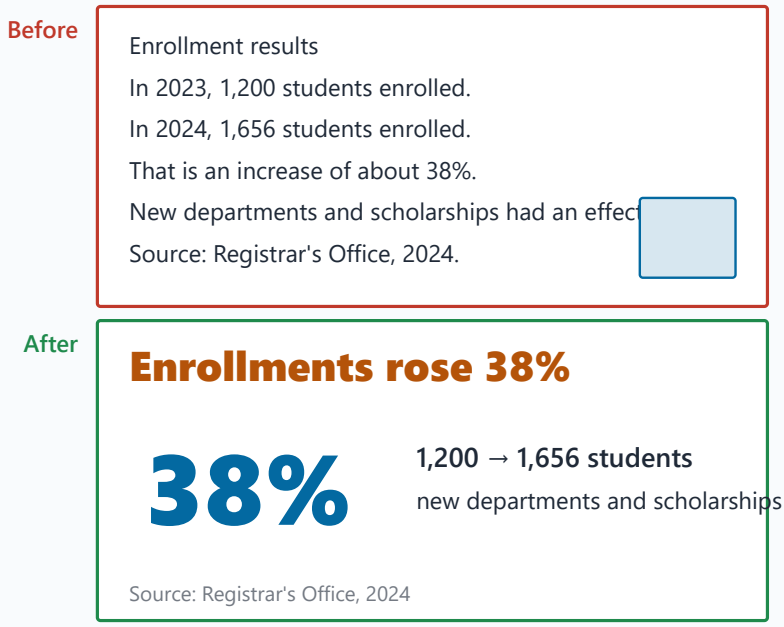


Figure 3.3.12: Visual hierarchy. At the top the title, the finding, and the source have the same size and weight. At the bottom the title states the result, 38% is the largest element, the detail is small, and the source is faint.

Example 3.3.6

In the slide in Figure 3.3.12, 1,200 students enrolled in 2023 and 1,656 in 2024. Let us find the rate of increase and rearrange the slide.

1. Amount of increase: 1,656 − 1,200 = 456 students.
2. Rate of increase: 456 ÷ 1,200 = 0.38, that is, 38%.
3. This rate is the main finding of the slide. The title states the result, not the topic: instead of “Enrollment results”, “Enrollments rose 38%”.
4. 38% becomes the largest element on the slide. Below it, in small type, go 1,200 → 1,656 students and a short reason (new departments and scholarships).
5. The source (Registrar's Office, 2024) is written smallest and faintest; it is read when needed but does not draw attention.

In the previous slide the five sentences have the same weight. The audience has to read all of them to find the most important piece of information. In the revised slide the eye goes first to 38%; the rest is read if needed.

Visual hierarchy has three tools. The title states the topic or the main result directly. Size and color show the order of importance; they are not used for decoration. Space and alignment also group the elements that belong together and draw a path for the eye. Elements that stand close together are perceived as related; space is left between unrelated ones.

Common mistake

Making everything bold and colored to emphasize the important information. If everything is emphasized, nothing is emphasized. On a slide usually a single element is brought forward.

3.3.4.5 Text on a slide

A talk slide is not a document page. The audience cannot read a long text and listen to the speaker at the same time; they choose one of the two, and usually they choose reading. That is why the text on a slide is kept short (Figure 3.3.13).



Figure 3.3.13: On the left a “wall of text” full of paragraphs. On the right three short bullets (goal, method, result) and the speaker note into which the explanation that does not fit on the slide has been moved.

There are six principles for slide text:

- **Short phrases:** The slide holds the main point, not the whole speech. Key phrases are often enough instead of full sentences.
- **Readable size:** Think of the audience member at the very back of the hall. Text that cannot be read from the back row might as well not be on the slide.
- **Consistent language:** The items in a list are written with the same grammatical structure. If one reads “Collecting the data” and another “We analyzed”, the list looks disorderly.
- **Contrast:** The text color is clearly separated from the background. A projector shows colors fainter than a computer screen.
- **Line length:** A line is kept short enough for the eye to follow at a glance.
- **Speaker note:** An explanation that does not fit on the slide is not deleted but moved to the notes pane.

Common mistake

Shrinking the font when the text does not fit. PowerPoint can shrink overflowing text by itself; but small text does not solve the problem, it only hides it. If the text does not fit, it is either shortened or split over two slides.

3.3.5 Pictures, tables, and charts

The saying “a picture is worth a thousand words” is largely true for talks, but only for the right picture. A visual that does not explain the topic is nothing more than a decoration taking up space on the slide. This section explains when and how to use pictures, shapes, tables, and charts.

3.3.5.1 Using pictures

A picture is added to a slide with Insert → Pictures. When a picture is selected, the Picture Format contextual tab opens; the crop, size, compress, and alt text commands are there. Six points are watched when using a picture:

- **Purpose:** The visual explains the topic or proves a claim. A photo of the measurement setup explains the method; a random stock photo of “a person working at a computer” explains nothing.
- **Resolution:** The picture has enough detail to look sharp on the presentation screen. A very small picture turns blurry when enlarged.
- **Cropping:** Brings the important region forward. When cropping, the ratio is kept; the picture is not stretched by pulling in only one direction.
- **Alt text:** Tells an audience member who uses a screen reader what the visual is about.
- **Source:** The license and attribution terms of someone else's visual are checked.
- **Consistency:** Similar visuals are used with the same ratio and placement.

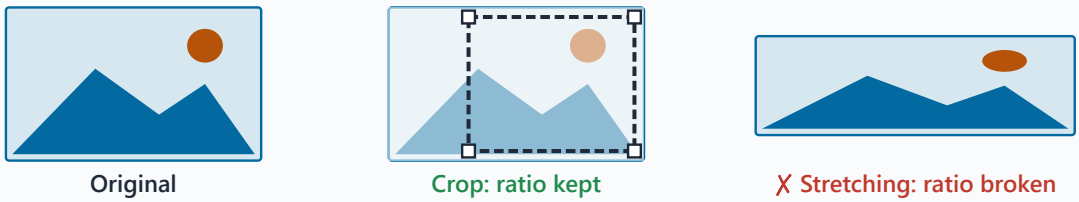


Figure 3.3.14: Three states of the same picture. On the left the original. In the middle, cropped with the ratio kept. On the right, merely stretched in the horizontal direction; the mountains have flattened and the sun has become an ellipse.

Definition 3.3.12

The **aspect ratio** is the ratio of a picture's width to its height. A 4000 × 3000-pixel photo has the ratio 4000 ÷ 3000 = 4 : 3, and a 1920 × 1080-pixel screen has 16 : 9. If the ratio is not kept when a picture is resized, the image flattens or stretches.

Example 3.3.7

A 4000 × 3000-pixel (4 : 3) photo is to fill a 16 : 9 slide exactly. How is the photo brought to this ratio without stretching, and how much of the picture remains?

1. The width is kept: 4000 pixels. At the ratio 16 : 9 the height must be 4000 × 9 ÷ 16 = 2250 pixels.
2. 3000 − 2250 = 750 pixels are cropped from the height. Depending on the subject, these rows are cut from the top, from the bottom, or shared between the two.
3. The remaining pixel count is 4000 × 2250 = 9,000,000. Of the original 4000 × 3000 = 12,000,000 pixels, 9,000,000 ÷ 12,000,000 = 0.75, that is, 75%, remains.

It is also possible to reach the same result by stretching, that is, by pulling the 4 : 3 picture horizontally by a factor of 16/9 ÷ 4/3 = 4/3 ≈ 1.33. But then every circle in the picture turns into an ellipse and every square into a rectangle (Figure 3.3.14).

Common mistake

Enlarging a picture by pulling the handle in the middle of an edge. A handle in the middle of an edge changes only one dimension and breaks the ratio. To keep the ratio, a corner handle is used or Shift is held down.

3.3.5.2 Picture size and compression

Even phones now take photos above 12 megapixels. If these photos are added to a slide as they are, the deck file quickly reaches hundreds of megabytes. Yet the screen can show only a fraction of these pixels.

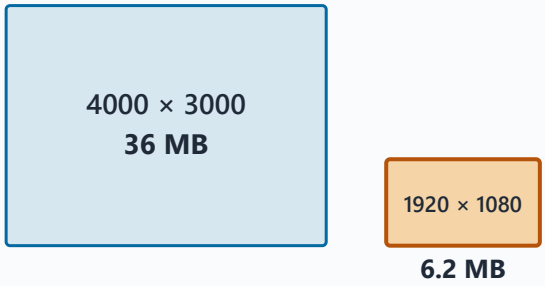


Figure 3.3.15: A size comparison of a 4000 × 3000-pixel photo and a 1920 × 1080-pixel screen. Uncompressed, the photo is 36 MB, and the image that fills the screen is about 6.2 MB.

Example 3.3.8

A 4000 × 3000-pixel photo will be shown full screen on a 16 : 9 slide. The projector gives a 1920 × 1080-pixel image. If every pixel is stored uncompressed in 3 bytes (one byte each for red, green, and blue), how much of the photo is unnecessary?

1. Raw size of the photo: 4000 × 3000 × 3 = 36,000,000 bytes, that is, 36 MB.
2. Raw size of the image that fills the screen: 1920 × 1080 × 3 = 6,220,800 bytes, that is, about 6.2 MB.
3. Ratio: 36,000,000 ÷ 6,220,800 ≈ 5.8. The photo carries about 5.8 times more data than the screen can show.

The extra pixels are not visible on screen; they only enlarge the file, slow down opening and saving, and make sending by e-mail harder. In a real file the pictures are stored compressed as JPEG or PNG, so the sizes are smaller. But the ratio is roughly preserved.

Definition 3.3.13

ppi (pixels per inch) shows how many pixels fall on a length of one inch (2.54 cm) of a picture. When the physical size of the picture on the slide is fixed, if ppi rises, the number of pixels, and therefore the file size, rises too.

The Picture Format → Compress Pictures command reduces the picture to a target ppi value. A value suitable for the intended use is chosen: for example 220 ppi for print, 150 ppi for web and screen, and 96 ppi for e-mail. In the same window there is also the option “Delete cropped areas of pictures”.

Example 3.3.9

A 16 : 9 slide measures 13.333 × 7.5 inches (33.867 × 19.05 cm). If a picture that completely fills the slide is compressed to different ppi values, how many pixels does it become?

Target	ppi	Width × height	Raw size
Print	220	2933 × 1650	≈ 14.5 MB
Web and screen	150	2000 × 1125	6.75 MB
E-mail	96	1280 × 720	≈ 2.8 MB

Table 3.3.12: Pixel size and raw size of a full-slide picture (3 bytes per pixel).

The calculation is the same on every row: width 13.333 × ppi, height 7.5 × ppi. For example, at 150 ppi, 13.333 × 150 ≈ 2000 and 7.5 × 150 = 1125 pixels are found. This value covers a 1920 × 1080 projection. When a 4000 × 3000 photo is compressed to 150 ppi, nothing is lost on screen, but the file gets noticeably smaller.

There is an important relationship between cropping and compression. The part hidden with Picture Format → Crop is actually not deleted; it stays in the file and the crop can be undone at any time. If "delete cropped areas" is not chosen when compressing, the file size does not shrink. Picture Format → Remove Background, on the other hand, makes the picture's background transparent; the subject sits directly on the slide's color.

Common mistake

Thinking the file shrinks when a photo is cropped. The cropped area stays in the file until it is deleted during compression. After the cropped areas are deleted, the crop can no longer be undone; this step is done after the deck is finished.

3.3.5.3 Shapes, icons, and SmartArt

Not every visual is a photograph. A process, a ranking, or a relationship between two things is often shown better with simple shapes. PowerPoint has several tools for this (Table 3.3.13).

Tool	What for?
Shape	Showing simple emphasis, a process, and relationships (Insert → Shapes)
Icon	Representing a familiar concept in a small space (Insert → Icons)
SmartArt	Visualizing a list or a process with a ready-made layout
Text box	Positioning limited text outside placeholders
Connector	A line or arrow that stays attached when shapes are moved

Table 3.3.13: Visual tools of the shape kind.

Definition 3.3.14

SmartArt is a tool that places text into a ready-made graphic layout such as a process, cycle, hierarchy, or list. The text stays as the SmartArt's content and remains editable; when the layout is changed, the boxes and arrows are redrawn automatically.

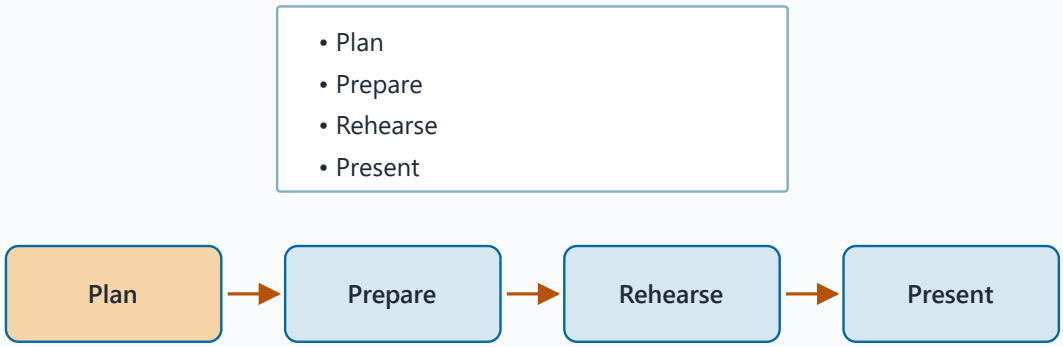


Figure 3.3.16: At the top four sequential steps are written as a plain bullet list. At the bottom, with Home → Convert to SmartArt → Basic Process, the same text has turned into boxes joined by arrows.

Figure 3.3.16 shows what a bullet list lacks: a plain list does not show order. When the items “Plan, Prepare, Rehearse, Present” are written one under another, it is understood that they form a process only after reading. When the same text is converted to SmartArt, the arrows give the sequence relationship at first glance. The text does not disappear; it moves into the SmartArt.

There is one more principle when working with shapes: color, line weight, and style are kept consistent across the whole deck. If on one slide a blue box means “input” and a red box means “output”, the same colors carry the same meaning on the other slides too.

Common mistake

Joining shapes to each other with plain lines. When a shape is moved, the plain line stays where it was and the connection breaks. A connector, on the other hand, attaches its ends to the connection points of the shapes; when a shape is moved, it lengthens or shortens along with it.

3.3.5.4 Tables

A table is the clearest way to compare numbers or attributes in rows and columns. But a table in a talk differs from a table in a report. The person reading a report can spend as much time as they like; the person watching a talk has a few seconds.

✗ Dense: 6 columns, small numbers

C1	C2	C3	C4	C5	C6
47	10	63	26	79	42
84	47	10	63	26	79
31	84	47	10	63	26
68	31	84	47	10	63
15	68	31	84	47	10

✓ Plain: only what the message needs

Year	Enrollment
2023	1,200
2024	1,656

Figure 3.3.17: On the left a dense table with six columns full of small numbers. On the right only the two rows the message needs: 1,200 enrollments in 2023 and 1,656 in 2024; the important cell is highlighted.

When preparing a table for a talk, the following are watched:

- The header row explains the meaning and unit of each column.
- Only the columns needed for the comparison are shown. If two of six columns carry the message, the other four are taken off the slide and, if needed, put on a backup slide.
- The important cell is made prominent with a restrained color or weight.
- Text is aligned left and numbers right. In right-aligned numbers the units, tens, and hundreds digits line up, and magnitudes can be compared at a glance.
- The source and date of the data are added in small type below the table when needed.
- If the table is still dense, a chart or a few key numbers are considered.

Common mistake

Pasting a table from Excel into a slide with all its rows and columns. The text size shrinks and the back rows cannot read a single number. A table on a slide is reduced to the few rows and columns that carry the message.

3.3.5.5 Charts

A chart turns the relationship between numbers into a shape. Which chart to choose depends not on the type of the data but on the message to be conveyed (Table 3.3.14).

Message	Chart	Example
Comparing categories	Bar or column	Student counts of the departments
Showing change over time	Line	Monthly electricity consumption
Showing a part's ratio to the whole	Pie with few categories, or a percent bar	Distribution of a budget over items
Showing the relationship between two variables	Scatter	Study hours and exam grade

Table 3.3.14: Chart type by message.

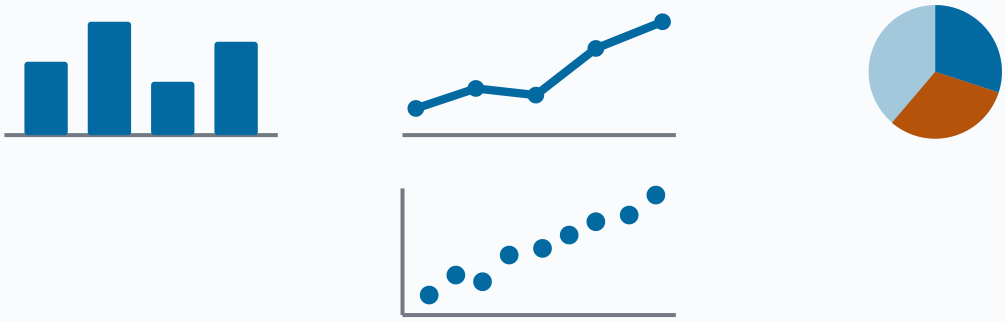


Figure 3.3.18: Small examples of column, line, pie, and scatter charts, from left to right.

Example 3.3.10

A student's monthly budget is divided into three items: rent 4,500 TL, food 3,000 TL, and transport 1,500 TL. For the message “Half of the budget goes to rent”, which chart is suitable, and what are the shares in the chart?

1. The message is a part-to-whole relationship and there are only three categories: a pie chart is suitable.
2. Total: 4,500 + 3,000 + 1,500 = 9,000 TL.
3. Shares: rent 4,500 ÷ 9,000 = 50%, food 3,000 ÷ 9,000 ≈ 33.3%, transport 1,500 ÷ 9,000 ≈ 16.7%.
4. The angles of the pie slices are the shares' equivalents in 360°: rent 0.5 × 360 = 180°, food 120°, transport 60°. The rent slice covers exactly half of the circle; the message is seen at first glance.
5. The chart title states the message: “Half of the budget goes to rent”.

If the number of items rises to seven or eight, the slices get thin and look alike. In that case a percent bar or a sorted bar chart is more legible.

Three rules hold for every chart type. The chart title states the main result plainly. Axes and units are visible. Three-dimensional effects are not used, because perspective makes the front bar or slice look bigger than it is and spoils the comparison.

Common mistake

Showing change over time with a pie chart. A pie chart shows the shares at a single moment; when the pies of two years are put side by side, the slices cannot be compared. Data with a time axis is given with a line chart.

3.3.5.6 Embedded and linked charts

Charts are often prepared in Excel and brought into the deck. There are three ways of bringing them in, and in each the place where the data sits is different (Table 3.3.15).

Definition 3.3.15

An **embedded object** is an object copied into the deck file together with its data; it is independent of the source file. A **linked object**, on the other hand, reads its data from the source file; the deck holds only the path of the source.

Type	How it arises	Where is the data?	When the Excel file changes
Embedded chart	Insert → Chart; or the option to embed the workbook when pasting	Inside the deck file	The chart does not change; the data is updated inside the deck with Edit Data
Linked chart	The option to link the data when pasting	In the source Excel file	The chart can be updated; if the file is moved or deleted, the link breaks
Picture	The option to paste as a picture	No data, only an image	Does not change; the chart can no longer be edited

Table 3.3.15: The three ways of bringing an Excel chart into a deck.

Example 3.3.11

Let us choose the suitable method for two situations.

1. The talk will be given on another computer in a conference hall, and only the .pptx file will be carried on a USB stick. The Excel file will not be with it. The chart's data must be inside the file: an **embedded chart**. There is no risk of a broken link.
2. A consumption report in the same layout is presented every month, and the figures are updated in Excel each month. Instead of copying the figures into the deck one by one each month, the chart is wanted to be fed from Excel: a **linked chart**. The deck and the Excel file are kept in the same folder and the folder is carried together.

Common mistake

Carrying a deck with a linked chart to another computer alone. The chart appears with its last saved image, but when an update is wanted PowerPoint cannot find the source file. Either the Excel file is carried as well or the chart is made embedded.

3.3.6 Alignment and layout

The audience may not consciously notice why two boxes with a difference of a few pixels between them do not stand in line. But they feel that the slide looks “messy”. Placing by eye is therefore insufficient. PowerPoint offers tools that align objects numerically.

3.3.6.1 Layout tools

Definition 3.3.16

Align brings the edges or centers of the selected objects to a common line: left, center, right, top, middle, bottom. **Distribute** leaves the two outermost objects in place and equalizes the gaps between them. Both are in the Shape Format → Align menu.

Tool	What it does
Align	Brings the edges or centers of objects to a common line
Distribute	Equalizes the space between objects
Guides and grid	Provide the same layout lines across a slide and across slides (View tab)
Group	Moves and resizes several objects as if they were one (Ctrl + G)
Bring Forward / Send Backward	Changes the order of overlapping objects
Selection Pane	Lists objects by name; manages visibility and layer order

Table 3.3.16: Layout tools.

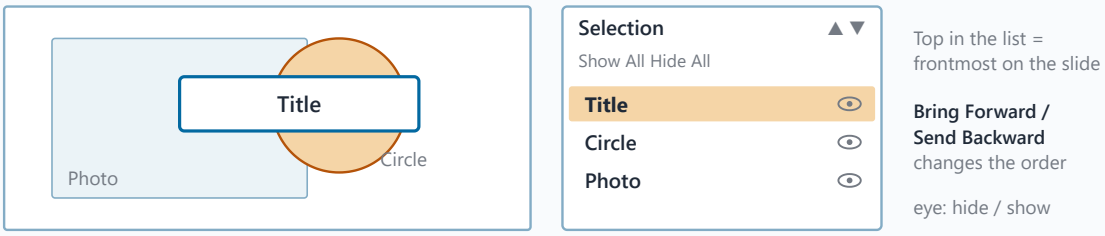


Figure 3.3.19: Overlapping Title, Circle, and Photo objects and the Selection Pane. The object at the top of the list is frontmost on the slide; the eye icon hides or shows an object.

The Selection Pane is especially useful on crowded slides (Figure 3.3.19). Selecting the rear one of overlapping objects with the mouse is hard; clicking its name in the pane is enough. The object higher in the list is in front on the slide. Bring Forward and Send Backward change the order in the list, and so the layer order on the slide. In the same pane objects can be given meaningful names. These names also appear in the Animation Pane; an animation list that says “Chart” instead of “Picture 7” is much easier to read.

Common mistake

Moving objects one by one after aligning and spoiling the layout. Objects that must stay together are grouped after being aligned. A group moves like a single object and the gaps inside it are preserved.

3.3.6.2 Align and distribute step by step

The align and distribute commands work numerically. So their results can be calculated in advance. Figure 3.3.20 and Table 3.3.17 follow three shapes step by step. The slide frame runs horizontally from 40 to 680; the positions of the shapes are given by the coordinates of their top-left corners.

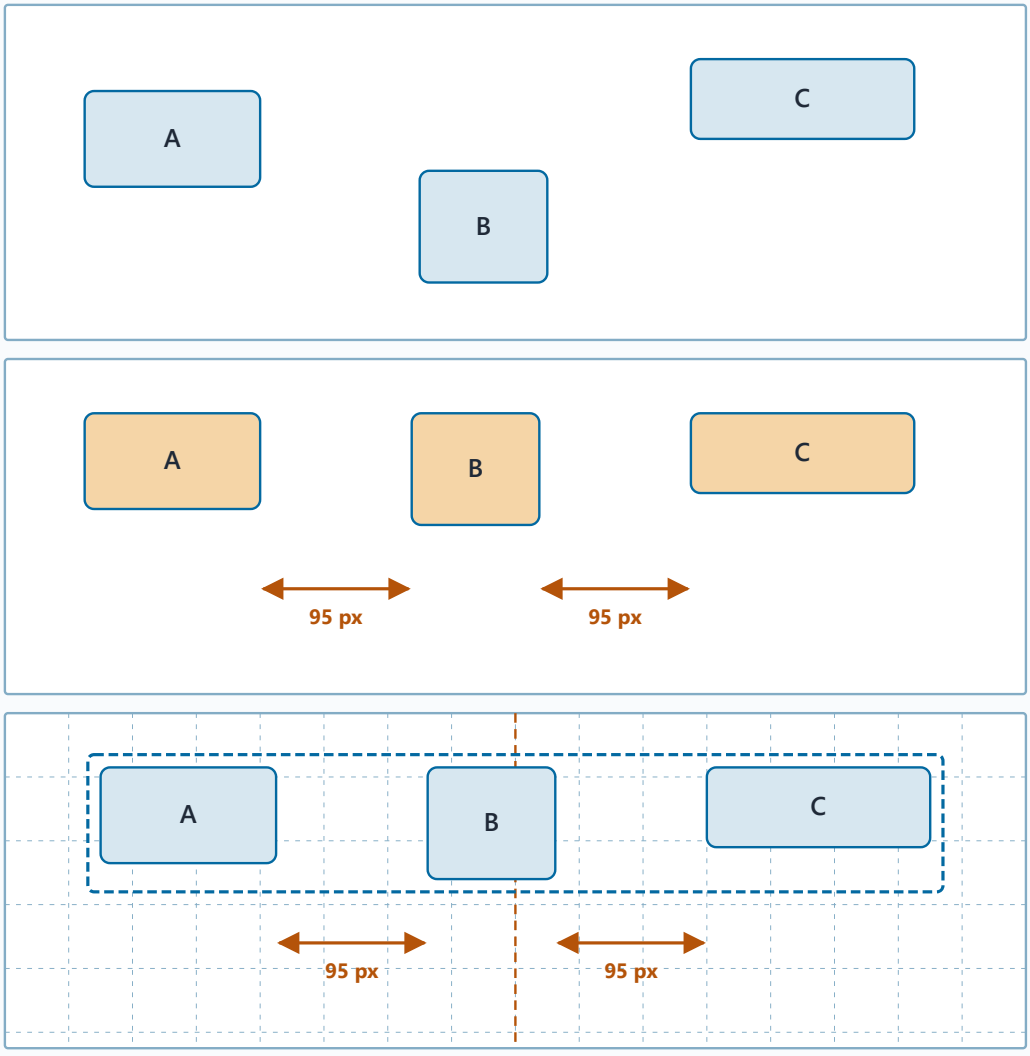


Figure 3.3.20: Align and distribute. At the top, three shapes placed by eye. In the middle, after Align Top and Distribute Horizontally, the top edges are on the same line and the gaps are 95 px. At the bottom the shapes have been grouped and centered on the slide.

Example 3.3.12

Initially there are the shapes A (x = 90, y = 60, width 110), B (x = 300, y = 110, width 80), and C (x = 470, y = 40, width 140).

Step	Command	A (x, y)	B (x, y)	C (x, y)
0	Placed by eye	90, 60	300, 110	470, 40
1	All three selected, Align → Align Top	90, 40	300, 40	470, 40
2	Align → Distribute Horizontally	90, 40	295, 40	470, 40
3	Group, Align → Align Center	100, 40	305, 40	480, 40

Table 3.3.17: Coordinates in the align, distribute, and group operations.

1. **Align Top:** The top edge of the topmost shape is C's (y = 40). The y values of A and B are pulled to 40; the x values do not change.
2. **Distribute Horizontally:** A, the leftmost, and C, the rightmost, stay in place. A's left edge is 90 and C's right edge is 470 + 140 = 610. The length of this span is 610 − 90 = 520. Because the sum of the shapes' widths is 110 + 80 + 140 = 330, the empty space is 520 − 330 = 190. Since the two gaps must be equal, each is 190 ÷ 2 = 95. B's new x value is 90 + 110 + 95 = 295. Check: B's right edge is 295 + 80 = 375, and its distance to C is 470 − 375 = 95.
3. **Group and center:** The group runs from 90 to 610; its center is (90 + 610) ÷ 2 = 350. The center of the slide frame is (40 + 680) ÷ 2 = 360. The group shifts 360 − 350 = 10 units to the right. Because every shape moves by the same amount, the gaps stay at 95.

It should be remembered that the Distribute command does not touch the outermost objects. If the distributed objects are to stand exactly in the middle of the slide, they are first distributed, then grouped and centered.

The guides and grid lines on the View tab carry these calculations across slides. For example, if on all the slides the content starts at the left on the same guide line, the text does not “jump” when the slide changes. The dashed line in the middle shows the slide's center.

Common mistake

Trying Distribute with only two objects selected. Because there is no intermediate object to distribute, nothing changes. Distribute gains meaning with at least three objects; two objects are only aligned.

3.3.7 Transitions and animation

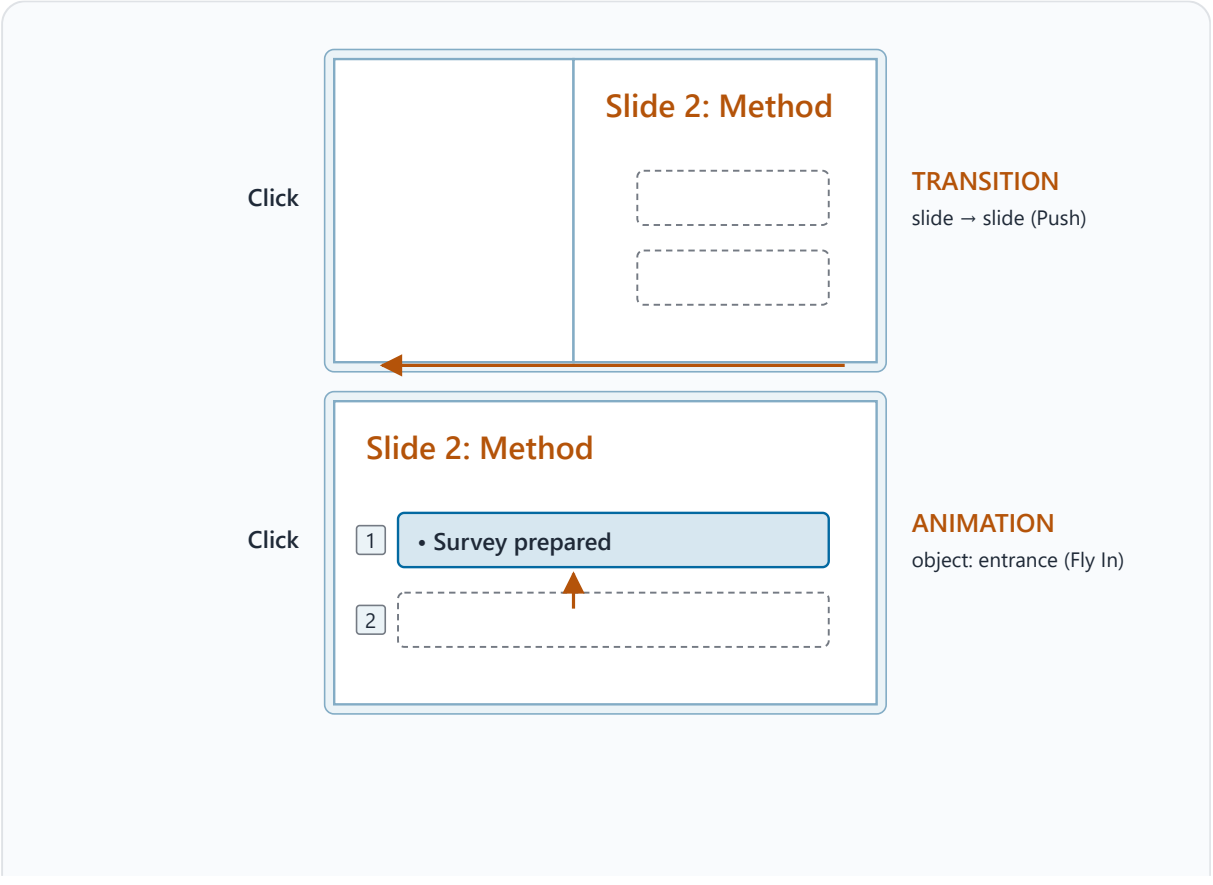
There are two separate kinds of motion in PowerPoint, and the two are often confused. One determines how the screen changes when moving from one slide to another; the other determines how objects within a slide appear or disappear. Used well, motion directs attention and opens up a process step by step. Used badly, it draws more attention than the content.

3.3.7.1 The difference between a transition and an animation

Definition 3.3.17

A **transition** is the way the screen changes when moving from one slide to the next, and it applies to the slide as a whole; it is given from the Transitions tab. An **animation** is the appearance, emphasis, disappearance, or movement of a single object within a slide (a title, a bullet, a picture, a chart); it is given from the Animations tab.

The best way to see the difference is to follow what each click does in the show. Figure 3.3.21 and Table 3.3.18 give a two-slide show click by click. Slide 2 has been given a Push transition. An entrance animation has been applied to the two bullets of Slide 2, and the first bullet has also been given an emphasis animation.



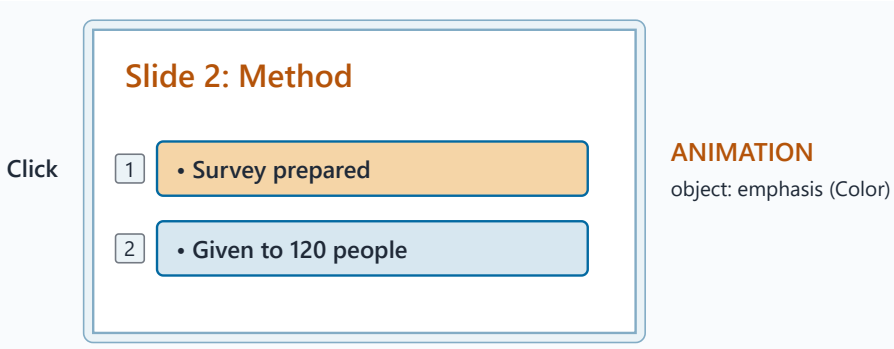


Figure 3.3.21: Transition versus animation. At the top, on a click Slide 2 comes in from the right and pushes Slide 1 out: the transition applies to the slide as a whole. In the middle, with the slide staying put, the first bullet appears with an entrance animation. At the bottom, an emphasis animation is applied to the already visible first bullet.

Click	What happens	Type	Slide on screen
Start	The show starts	—	Slide 1
1	Slide 2 comes in from the right and pushes Slide 1; the bullets are not visible yet	Transition (Push)	Slide 2
2	The “Survey prepared” bullet appears	Entrance animation	Slide 2
3	The “Given to 120 people” bullet appears	Entrance animation	Slide 2
4	The color of the first bullet changes	Emphasis animation	Slide 2
5	The show moves on to the next slide	Transition	Slide 3

Table 3.3.18: The effect of each click in a two-slide show.

In the table, the transition is seen only when the slide changes. Animations, on the other hand, run while the slide stays in place. Because Slide 2 has three animations, leaving Slide 2 takes four clicks: three for the animations and the fourth for the transition. PowerPoint's rule is this: a click does not move to the next slide until all the click-started animations of a slide have finished; it first starts the next animation.

Having the bullets come one at a time ties the delivery to the audience's pace. The audience does not read a bullet that has not yet been spoken about; their attention stays on the point the speaker is making at that moment. The small numbers that appear next to objects in the editing view show this order. These numbers are not visible in the show.

Common mistake

Putting each bullet on a separate slide and giving a transition to make the bullets come one at a time. This increases the slide count needlessly, and the whole screen changes at every click. Making the bullets on the same slide come in order is the job of the animation.

3.3.7.2 Transitions

A transition is given to the selected slide from the Transitions tab and plays as that slide comes onto the screen. Figure 3.3.22 shows three common transition types over time.

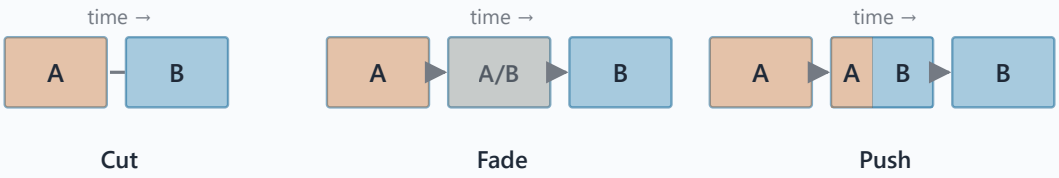


Figure 3.3.22: Three transition types. In Cut, A turns into B at once. In Fade, A and B are seen overlapping for a short time. In Push, B pushes A out of the screen as it comes in.

Setting	What it determines
Effect	How the slide comes onto the screen
Duration	How many seconds the transition lasts
Effect Options	The direction or variant (e.g., whether Push comes from the right or from the bottom)
Apply To All	Giving the selected transition and its duration to all slides
Advance Slide	Whether to move to the next slide on click or after a set time

Table 3.3.19: Transition settings.

The main principle for transitions is consistency. Few kinds of transition are used in a deck; often a single plain transition is enough. If every slide is given a different transition, the audience waits at each slide change to see “what will it be this time” and attention slips from the content. A more noticeable transition can be used only to mark a turning point in the talk, for example the start of a new section.

Common mistake

Making the transition duration long. A two-second transition in a 30-slide talk spends a whole minute merely on screen changes. The transition duration is kept short; the audience focuses on the content without noticing the transition.

3.3.7.3 Designer, Morph, and Copilot

In current versions of PowerPoint there are three tools that make design and motion easier. The presence and names of these tools may vary with the PowerPoint version, the license, and the update level; tab and button names may look slightly different in the version in use.

- **Designer**: Offers ready-made layout suggestions by looking at the content on the slide; it is on the Design tab. Because the suggestions are generated online, it requires an internet connection and, in the desktop version, a Microsoft 365 subscription.
- **Morph**: A transition type on the Transitions tab. If the same object stands at a different position, size, or color on two consecutive slides, PowerPoint generates the motion in between by itself.
- **Copilot**: Offers AI assistance such as creating draft slides and summarizing the deck. It works with a monthly AI credit on a personal Microsoft 365 subscription and with a separate Copilot license on an organizational account.

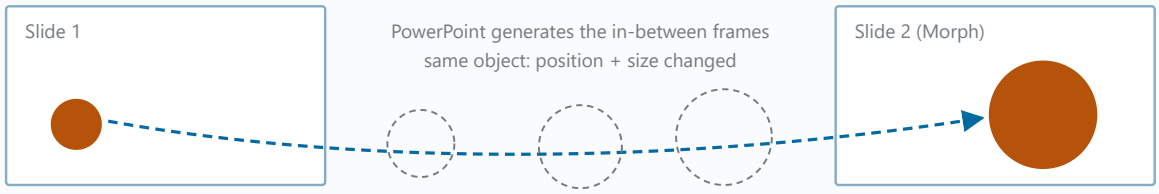


Figure 3.3.23: The Morph transition. The same circle is small and on the left in Slide 1, and large and on the right in Slide 2. When Morph is given to Slide 2, PowerPoint generates the frames in between; the circle grows as it slides to the right.

Definition 3.3.18

Morph is a transition that animates the motion in between automatically by finding the differences in position, size, rotation, and color of the objects common to two consecutive slides. The transition is given to the second slide.

The easy way to Morph is this: the first slide is duplicated, the objects on the copy are moved or resized to their new positions, and the Morph transition is given to the second slide. Because the slide was duplicated, the objects on the two slides are the "same" objects as far as PowerPoint is concerned. If the objects were drawn separately, PowerPoint may not be able to match them; in that case one fades out and the other fades in.

Common mistake

Accepting a layout from the Designer pane without thinking. A suggestion may look good aesthetically but spoil the visual hierarchy: for example, it may shrink the main number and enlarge a decorative picture. A suggestion is used if it brings out the slide's single message.

3.3.7.4 Animation types

Animations fall into four types (Figure 3.3.24). In the gallery on the Animations tab each type is listed in a separate group, and their icons have different colors.

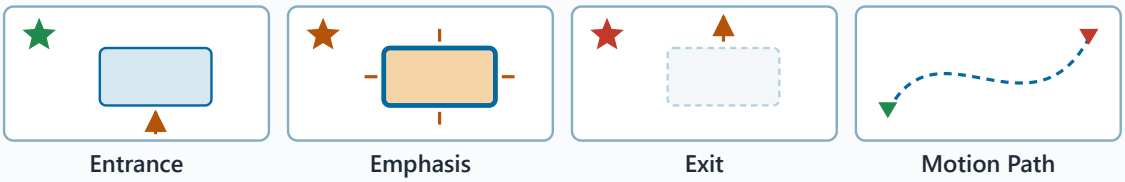


Figure 3.3.24: Animation types. Entrance brings the object onto the slide; emphasis draws attention to an already visible object; exit removes the object from the slide; a motion path moves the object along the route shown by a dashed line.

Type or tool	What it does	Example
Entrance	Makes the object appear on the slide	A bullet flying in from the left
Emphasis	Draws attention to an object that is already visible	A number changing color
Exit	Removes the object from the slide	The old value fading away
Motion Path	Moves the object along a particular route	A packet moving along a network diagram
Trigger	Starts the animation when a particular object is clicked	An answer appearing when a button is clicked
Animation Pane	Shows and manages the order, start, duration, and delay of all effects	—

Table 3.3.20: Animation types and animation tools.

An object can be given more than one animation. For this Animations → Add Animation is used. Choosing an effect directly from the gallery replaces the object's existing effect with the new one. For example, if a bullet will first come in with an entrance and then be emphasized, the emphasis is added with Add Animation.

Common mistake

Giving an entrance animation to draw attention to a visible object. An object with an entrance animation is not visible when the slide opens; it appears only when its animation runs. For an object that must already be visible, the emphasis type is chosen.

3.3.7.5 Start, duration, and delay

Every animation has three timing settings: when it will start, how long it will last, and how long it will wait before starting. These settings are in the Timing group of the Animations tab or in the Animation Pane.

Definition 3.3.19

An animation's **start** can be of three kinds. **On Click**: starts when the presenter clicks or presses a key. **With Previous**: starts at the same time as the previous effect in the list. **After Previous**: starts by itself when the previous effect finishes. The **duration** determines how long the effect lasts, and the **delay** how long the effect waits after the start moment.

From these settings the start and end moments of an effect are calculated with a simple rule:

- Reference point: for On Click the moment of the click, for With Previous the start of the previous effect, for After Previous the moment the previous effect(s) finish.
- Start = reference point + delay.
- End = start + duration.

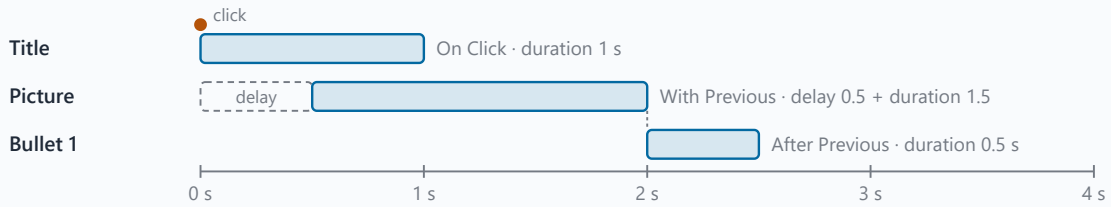


Figure 3.3.25: Timeline. The Title starts on click and lasts 1 s. The Picture plays With Previous, with a 0.5 s delay and a 1.5 s duration, between 0.5 and 2 s. Bullet 1 plays After Previous, with a 0.5 s duration, between 2 and 2.5 s.

Example 3.3.13

Let us calculate the times of the three effects in Figure 3.3.25 by the rule. The presenter clicks at t = 0.

1. **Title** (On Click, delay 0, duration 1 s): the reference point is the click moment, that is, 0. Start 0 + 0 = 0, end 0 + 1 = 1 s.
2. **Picture** (With Previous, delay 0.5, duration 1.5 s): the reference point is the Title's start, that is, 0. Start 0 + 0.5 = 0.5 s, end 0.5 + 1.5 = 2 s.
3. **Bullet 1** (After Previous, delay 0, duration 0.5 s): the previous effect that finishes last is the Picture (2 s). Start 2 s, end 2 + 0.5 = 2.5 s.

The three effects are completed in 2.5 seconds with a single click. The Title and the Picture play at the same time between 0.5 and 1 s.

Animations are always rehearsed in show view. Effects that look independent of one another on the editing screen may overlap unexpectedly in the show. For accessibility, too, fast-blinking and needlessly repeating motion is avoided. Such motion causes discomfort and distraction in some viewers.

Common mistake

Taking the delay to mean "waiting after the previous effect finishes". The delay of an effect that starts With Previous is counted from the start of the previous effect, not from its end. The Picture's 0.5 s delay elapses 0.5 s after the Title starts, not after it ends.

3.3.7.6 The animation timeline

In animations spread over several clicks, the numbers in the Animation Pane gain importance. The number next to an effect in the pane shows with which click that effect starts. Effects that start With Previous and After Previous do not get their own number; they join the group of the click they depend on.

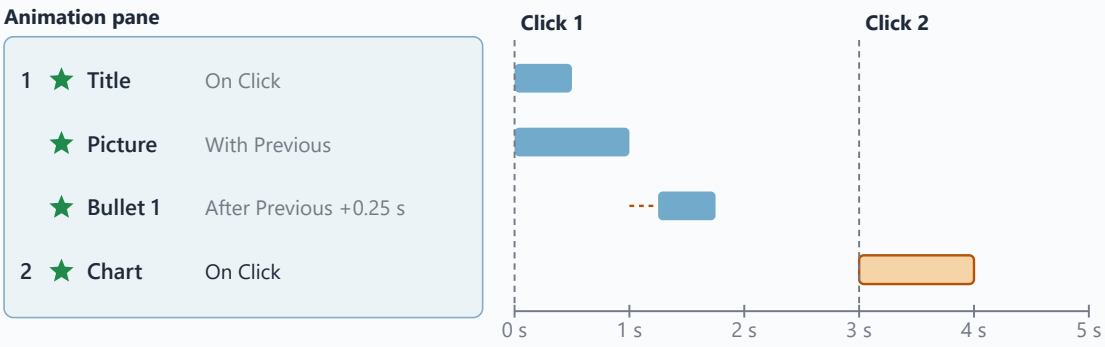


Figure 3.3.26: The Animation Pane and timeline. Click number 1 starts the Title, the Picture, and Bullet 1. The Chart plays with click number 2, the presenter's click at the 3rd second, between 3 and 4 s.

Example 3.3.14

There are four effects in the pane in Figure 3.3.26. The presenter clicks at t = 0 and at t = 3. What are the start and end moments of each effect?

Effect	Start type	Delay	Duration	Reference point	Interval played
Title	On Click (1)	0	0.5 s	Click 1: 0	0–0.5 s
Picture	With Previous	0	1 s	Title's start: 0	0–1 s

Effect	Start type	Delay	Duration	Reference point	Interval played
Bullet 1	After Previous	0.25 s	0.5 s	End of the previous effects: 1	1.25–1.75 s
Chart	On Click (2)	0	1 s	Click 2: 3	3–4 s

Table 3.3.21: The timeline of four effects.

1. The Title and the Picture start at the same time; the Title ends at 0.5, the Picture at 1.
2. Bullet 1 waits until the moment the previous effects have all finished: although the Title ends at 0.5, because the Picture ends at 1, the reference point is 1. A delay of 0.25 s is added on top: it starts at 1.25 s, lasts 0.5 s, and ends at 1.75 s.
3. The automatic chain ends at 1.75 s. Because the next effect starts On Click, PowerPoint waits for the presenter's click. Between 1.75 and 3 s nothing moves on the screen.
4. The presenter clicks at t = 3; the Chart starts at 3, lasts 1 s, and ends at 4.

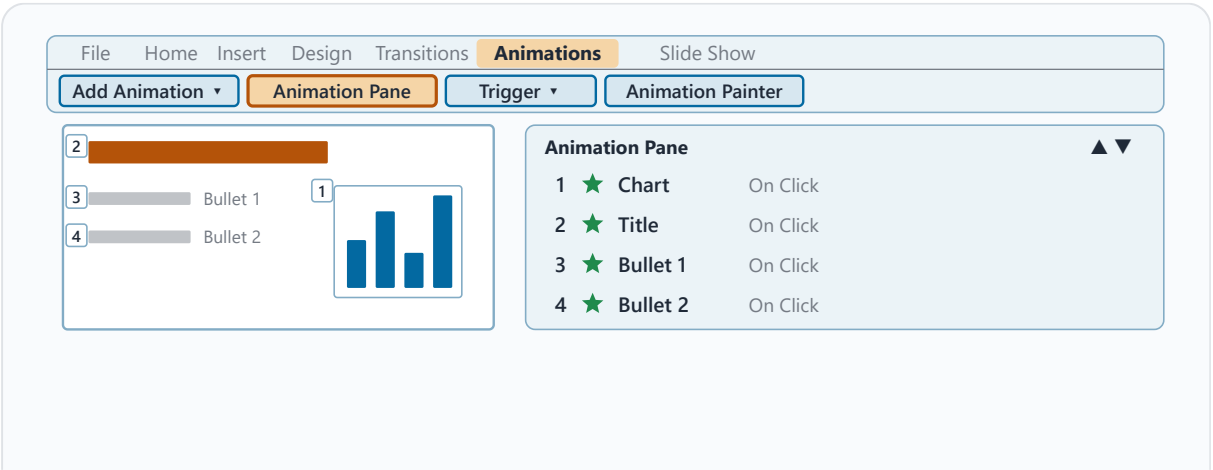
The numbers in the pane are only 1 and 2. Two clicks are enough for all the animations of this slide; the third click moves to the next slide.

Common mistake

Reading After Previous as “when the effect immediately above it in the list finishes” and looking at the wrong effect. The effect above can finish before a longer effect that started earlier. In the Bullet 1 example the one immediately above is the Picture and gives the right result; but had the Picture lasted 0.3 s, Bullet 1 would still have waited for the Title's end at 0.5.

3.3.7.7 Editing the animation order

Effects are listed in the Animation Pane in the order they were added. If the chart was animated first and then the title, in the show the chart comes before the title. The order is changed with the ▲ and ▼ buttons in the pane or by dragging the effect.



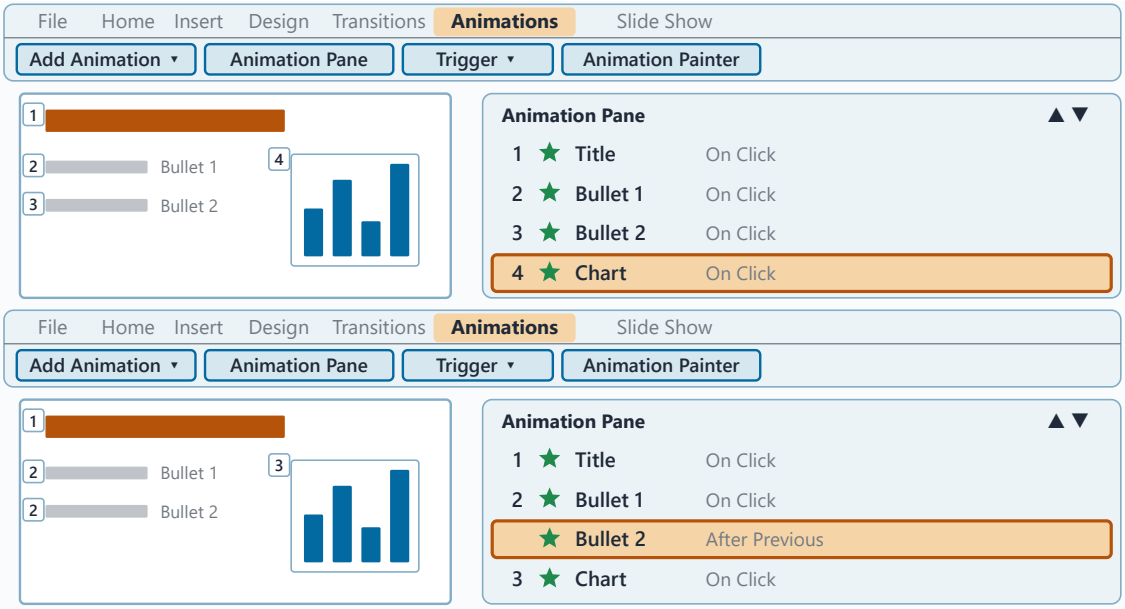


Figure 3.3.27: Editing the animation order. At the top the Chart is by mistake first in the order. In the middle, with ▲ and ▼, the order has become Title, Bullet 1, Bullet 2, Chart. At the bottom Bullet 2's start has been set to After Previous; Bullet 2 has lost its own number and shares the number (2) with Bullet 1.

Example 3.3.15

Step	Action	Order in the pane (number)	Click count
1	Start	Chart (1), Title (2), Bullet 1 (3), Bullet 2 (4)	4
2	Title is selected, ▲	Title (1), Chart (2), Bullet 1 (3), Bullet 2 (4)	4
3	Chart is selected, ▼ twice	Title (1), Bullet 1 (2), Bullet 2 (3), Chart (4)	4
4	Bullet 2's start is set to After Previous	Title (1), Bullet 1 (2), Bullet 2 (—), Chart (3)	3

Table 3.3.22: Reordering in the Animation Pane and the click count.

The first three steps change only the order; because each effect starts with its own click, the click count stays at 4. The numbers are reassigned at every step. In the fourth step Bullet 2 gets no separate click; it comes by itself when Bullet 1 finishes. Three clicks are enough in the show: Click 1 brings the Title, Click 2 brings Bullet 1 and then Bullet 2, and Click 3 brings the Chart. The fourth click moves to the next slide.

Common mistake

Deleting the animations and re-adding them in the right order to fix the order. This also deletes the duration, delay, and effect options. The order is changed with ▲ and ▼ in the Animation Pane; the settings of the effects are kept.

3.3.8 Delivering the talk

After the deck is prepared the real work begins: speaking in front of the audience. At this stage PowerPoint does more for the speaker than show slides. It gathers the notes, the timer, and the next slide on a screen only the speaker sees, lets the show be run from the keyboard, and measures the rehearsal.

3.3.8.1 Speaker notes

Definition 3.3.20

Speaker notes are text written in the notes pane, attached to a slide, that is only for the speaker. They are not visible to the audience in a normal show; they are shown to the speaker in Presenter View and can be printed as Notes Pages.

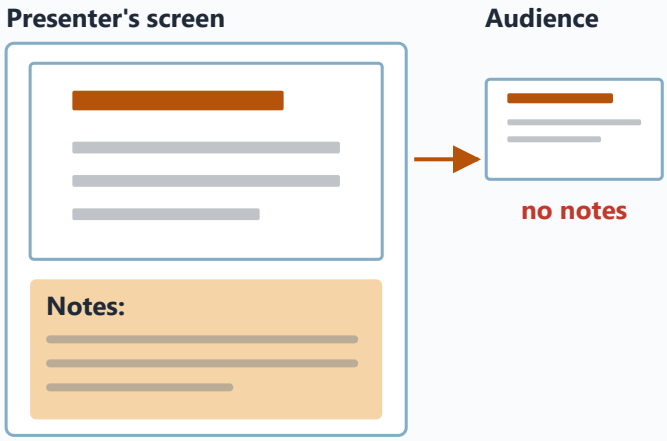


Figure 3.3.28: On the speaker's screen the notes are below the slide. On the projector the audience sees only the slide.

Four kinds of information are written in the notes:

- **Content:** Detail of the delivery that is not on the slide. The explanation behind the “38% increase” text on the slide stays in the note.
- **Reminder:** An example to give, a question to ask the audience, or a source.
- **Time:** The time set aside for this part, for example “The Method part should end here: minute 5”.
- **Transition sentence:** The sentence that connects to the next topic. Whether the flow breaks between two slides depends largely on this sentence.

Notes are also the safeguard for shortening the text on the slide. A sentence taken off the slide is not lost; it stays in front of the speaker's eyes. A paper copy of the notes can also be made with File → Print → Notes Pages.

Common mistake

Reading the notes sentence by sentence. A note is a reminder, not a script. A note written in short keywords is read at a glance; a long paragraph locks the speaker to the screen and breaks eye contact with the audience.

3.3.8.2 Presenter View

When a laptop is connected to a projector or a second screen, two screens exist. When the show starts, PowerPoint sends different things to these two screens: only the current slide to the projector, and Presenter View to the laptop screen.

Definition 3.3.21

Presenter View is the show view that appears only on the speaker's screen. It shows the current slide, the next slide, the speaker notes, the elapsed time, the clock, and the show tools (pen, laser pointer, zoom, all slides) together.

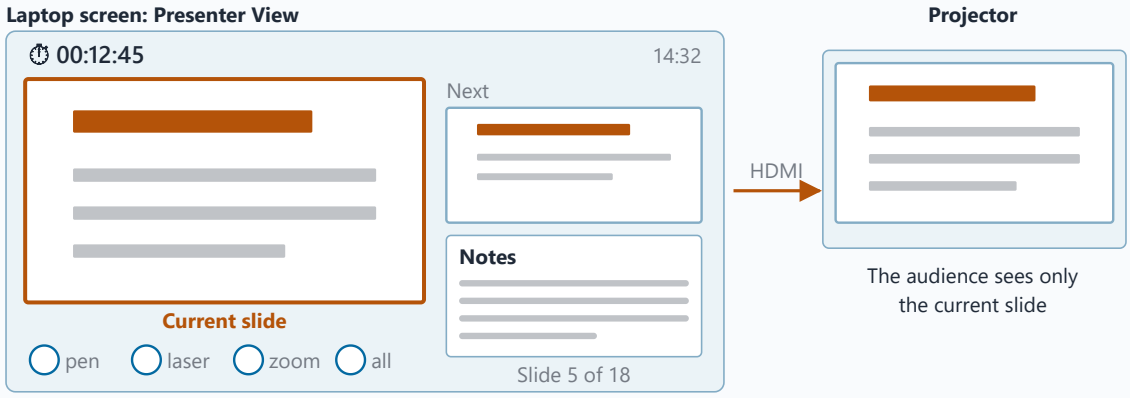


Figure 3.3.29: Presenter View on the laptop screen: on the left the current slide with the tools below it, on the right the next slide and the notes, at the top the timer (00:12:45) and the clock. On the projector connected with HDMI the audience sees only the current slide.

The tools of Presenter View keep the talk flowing (Figure 3.3.29):

- The **timer** shows the time elapsed since the show began; it can be paused and reset. Beside it is the real clock too.
- The **next-slide** preview lets the speaker prepare the transition sentence to the next topic.

- The **pen and laser pointer** are used to point at a particular area on the slide or to draw a short mark on it.
- The **All Slides** button jumps to a slide chosen from the thumbnails. The audience does not see this search screen; the current slide stays on the projector.

Presenter View opens by itself when the show starts with a second screen connected. Alt + F5 starts the show directly with Presenter View. This shortcut can also be used when rehearsing on a single screen.

Common mistake

Projecting Presenter View onto the projector when the screens get mixed up. In that case the audience sees the speaker's notes and the next slide. The positions of the two screens are swapped from the display settings at the top of Presenter View. This setting is tried out in the hall before the talk.

3.3.8.3 Show shortcuts

Opening menus with the mouse during the show both takes time and distracts the audience. The show is run with a few keys on the keyboard (Table 3.3.23).

Action	Key
Start the show from the beginning	F5
Start the show from the current slide	Shift + F5
Start with Presenter View	Alt + F5
Next animation or slide	→, Space, or Enter
Previous animation or slide	←
Blacken the screen / restore	B or . (period)
Whiten the screen / restore	W or , (comma)
Go to a particular slide	Slide number + Enter
Pen / laser pointer	Ctrl + P / Ctrl + L
End the show	Esc

Table 3.3.23: Slide show shortcuts.

Example 3.3.16

In the show of a 12-slide deck without animations, slide 7 is on the screen. The presenter presses the following keys in order. What is on the screen after each key?

Step	Key	Screen	Description
1	W	White screen	Attention is drawn to the speaker or the board

Step	Key	Screen	Description
2	W	Slide 7	Returns to the slide it was on
3	1, 1, Enter	Slide 11	A number and Enter jump straight to that slide
4	←	Slide 10	Previous slide
5	Esc	Normal view, Slide 10 selected	The show ends
6	Shift + F5	Slide 10	The show starts from the current slide
7	Esc, then F5	Slide 1	F5 always starts from the beginning

Table 3.3.24: A trace of show shortcuts.

In the third step the keys “1” and “1” are pressed in turn and then Enter; the number 11 is not typed on the screen, but PowerPoint collects the number. This is the fastest way to return to a particular slide during question time; that is why it helps for the speaker to keep slide numbers in the notes.

Common mistake

Pressing Esc in the middle of the show and starting again with F5. F5 starts the show from slide 1; the speaker has to go through all the slides again to reach the place where they left off. To continue from where you were, Shift + F5 is used.

3.3.8.4 Slide size, footer, audio, and video

Some of the technical settings of a talk must be made before the deck is prepared. The first of these is the slide size.

Definition 3.3.22

The **slide size** is the aspect ratio and physical measure of the slide; it is chosen with Design → Slide Size. Widescreen (16 : 9) is 33.867 × 19.05 cm, and standard (4 : 3) is 25.4 × 19.05 cm. Most of today's screens and projectors are 16 : 9; 4 : 3 is for old projectors.

Example 3.3.17

A deck prepared as 16 : 9 is later converted to 4 : 3, and PowerPoint shrinks the content to fit. To how many points does a 32-point title fall?

1. The two sizes have the same height (19.05 cm); the width falls from 33.867 cm to 25.4 cm.
2. For the content to fit the new width, the scale ratio is 25.4 ÷ 33.867 ≈ 0.75.

3. All the content shrinks by this ratio: 32 × 0.75 = 24 points. Pictures and shapes also shrink by a quarter; empty bands are left above and below the slide.

The other option is to place the content in the new size without shrinking it; in that case the objects on the right and left may spill off the slide. In both options the layout is spoiled and every slide has to be gone over. That is why the slide size is chosen before starting the deck.

The other technical settings are these:

- **Footer and slide number:** With Insert → Header & Footer, the slide number, date, and footer text are added; with Apply to All they spread to all slides. The slide number lets an audience member point at a slide in question time, saying “the chart on slide 12”.
- **Audio and video:** Added with Insert → Video or Insert → Audio. From the Playback tab that opens when the inserted media is selected, the way it starts (on click or automatically), trimming, and volume are set.
- **Font embedding:** File → Options → Save → Embed fonts in the file. The fonts used in the deck are put inside the file; when the deck is opened on a computer where that font is not installed, the text does not get garbled. The license of some fonts does not allow embedding.
- **Duplicating:** Ctrl + D copies the selected slide or object right next to itself.
- **Pen and laser:** During the show Ctrl + P opens the pen and Ctrl + L the laser pointer.

Common mistake

Trusting a font that looks fine on the computer where the deck was prepared. If the computer in the hall does not have that font, PowerPoint puts another font in its place; lines get longer and the text spills out of the boxes. The font is embedded or tried out on the hall computer before the talk.

3.3.8.5 Hidden slides and custom shows

The same deck can be presented to different audiences at different lengths. For example, a project is presented in 20 minutes to the jury and in 5 minutes on the department's open day. Instead of keeping a separate file for each, two tools are used.

Definition 3.3.23

A **hidden slide** is a slide that stays in the file but is skipped in the normal show. A **custom show** is a separate, named show sequence chosen from the slides of the same deck; it is created and played with Slide Show → Custom Slide Show.

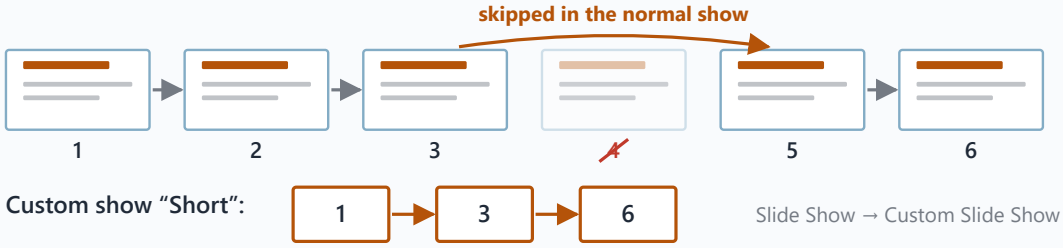


Figure 3.3.30: In a six-slide deck, slide 4 is hidden and is skipped in the normal show. The custom show named “Short” contains only slides 1, 3, and 6.

Example 3.3.18

Let us compare the three shows for the deck in Figure 3.3.30.

1. **Normal show with F5:** 1, 2, 3, 5, 6. The hidden slide 4 is skipped; five slides are shown.
2. **The “Short” custom show:** 1, 3, 6. A custom show uses its own list; slides 2 and 5, which are not in the list, are not shown either.
3. **If a question comes:** During the show 4 is typed and Enter is pressed. A hidden slide can be shown this way.

A hidden slide keeps backup detail ready for question time. A custom show produces talks of different lengths from the same file; when a slide is corrected, it is corrected in all the custom shows.

Common mistake

Hiding sensitive content and sharing the file. A hidden slide stays in the file; anyone who opens the file sees it in the slide pane. Sensitive content is deleted before sharing.

3.3.8.6 Rehearsing timings and a self-running show

A show does not always advance by the speaker's click. On an information screen or in a narrated recording the slides must advance by themselves at set times. The easiest way to determine these times is to rehearse the deck once at real speed.

Definition 3.3.24

Rehearse Timings is the Slide Show command that starts the show and measures and records the time spent on each slide. The recorded times are written into each slide's transition settings. A show of the **kiosk** type opens full screen, does not advance by clicking, and stays in a loop until Esc is pressed.

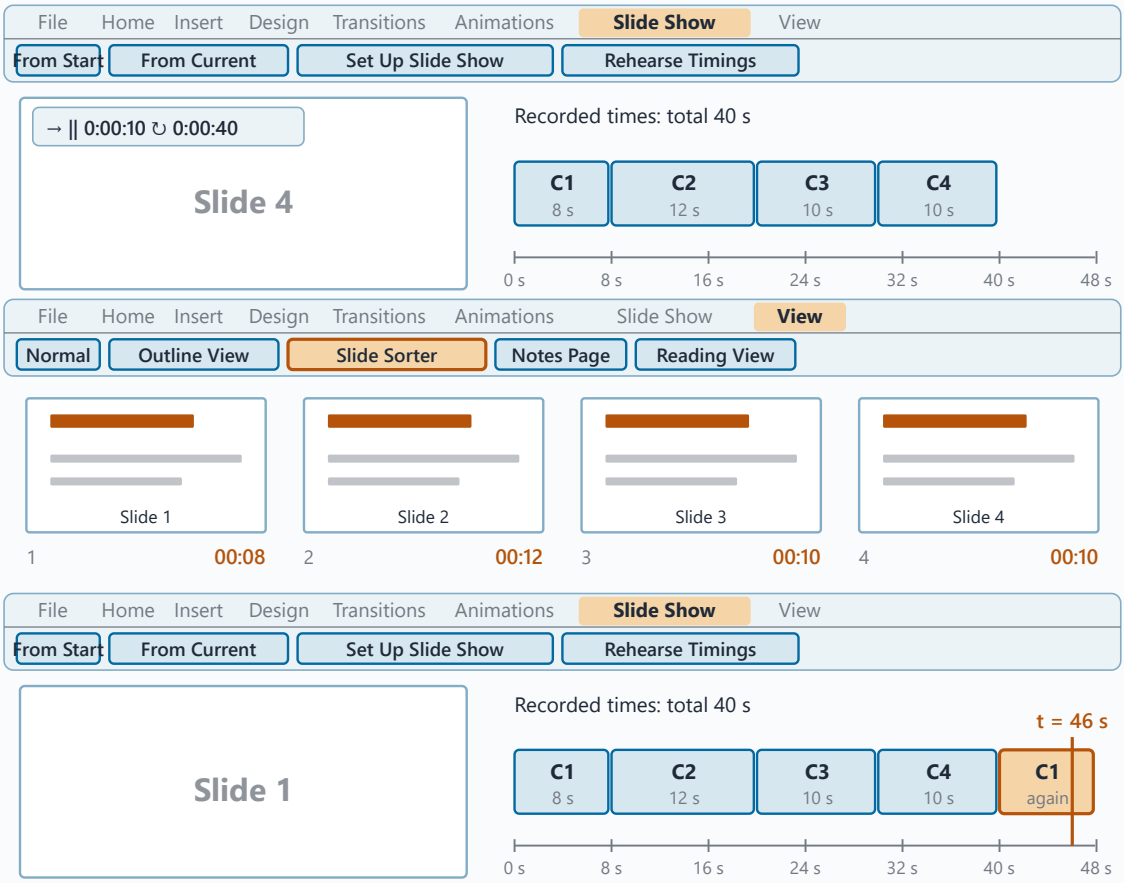


Figure 3.3.31: Timing rehearsal. At the top the times recorded for four slides: 8, 12, 10, and 10 s, a total of 40 s. In the middle the times appear under the thumbnails in Slide Sorter. At the bottom the kiosk show returns to the start after the last slide; at t = 46 s the screen shows Slide 1 again.

Step	Action	Result
1	Slide Show → Rehearse Timings	The show starts; the toolbar in the corner counts the slide time and the total time
2	Slide 1 is narrated at real speed and advanced	8 s is recorded for Slide 1
3	The other slides are narrated	12, 10, 10 s; total 8 + 12 + 10 + 10 = 40 s
4	The rehearsal ends and the timings are saved	Each slide's own time is written into the “After” box on its Transitions tab
5	Set Up Slide Show → kiosk type	Loop continuously is checked automatically; the show does not advance on click

Table 3.3.25: Steps of the timing rehearsal and the self-running show.

Example 3.3.19

A four-slide deck runs as a kiosk show (times 8, 12, 10, 10 s). What slide is on the screen t = 46 s and t = 75 s after the show starts?

1. One lap lasts 8 + 12 + 10 + 10 = 40 s. The slides' intervals within a lap: Slide 1 0–8, Slide 2 8–20, Slide 3 20–30, Slide 4 30–40 s.
2. t = 46: whole laps are discarded and the remainder is found: 46 − 40 = 6. The sixth second falls in Slide 1's interval (0–8). The screen shows Slide 1 of the second lap.
3. t = 75: 75 − 40 = 35. The 35th second falls in Slide 4's interval (30–40). The screen shows Slide 4; 35 − 30 = 5 s have passed in this slide.

The general rule: the remainder of t divided by the lap time is found and it is seen which slide's interval it falls in. For example in 10 minutes (600 s) the show makes 600 ÷ 40 = 15 laps.

Because clicking does nothing in a kiosk show, timing is essential. A kiosk show without timings gets stuck on the first slide. If the viewer is to navigate by touching the screen, only hyperlinks and action buttons work.

Common mistake

Forgetting the timings recorded during rehearsal in a live talk. The recorded times advance the slides without waiting for the speaker's click; if the speaker talks longer, the slide changes without waiting for them. Before a live talk, in Set Up Slide Show the option to advance manually is chosen.

3.3.8.7 Presenting effectively

Even the best-prepared deck is strengthened or weakened by the speaker's manner. There are six principles for the talk itself:

- The talk is rehearsed beforehand with the real screen and connections. The projector's resolution, its colors, and the connecting cable are tried before the day of the talk.
- The slide is not read out; eye contact and communication with the audience are kept. If the speaker turns to the slide and reads, the audience reads too and stops listening to the speaker.
- Every part is tied clearly to the main message. At the end of a part a linking sentence such as “this result tells us the following about our main question” is said.
- The pointer and animation are used only to direct attention. Drawing circles on the slide with the laser pointer distracts.
- The time limit is respected; room is left for questions and answers. If time is running over, the last slides are not rushed; a detail decided in advance is skipped.
- Alternatives such as a PDF backup, a screen adapter, and an offline copy are kept ready. If the computer, the software, or the internet connection causes trouble, the talk is continued from the PDF.

Common mistake

Rushing through slides when time runs short and saying “there's no time for these”. The audience takes the skipped slides to be important and thinks they are missing something. If time is not enough, the speaker jumps to the closing slide before reaching the end and states the main message.

3.3.9 Quality and sharing

When the deck is finished, the job is not considered done. Before sharing, it is checked that the deck can be read by everyone, carries no unwanted information, and works correctly on another computer.

3.3.9.1 An accessible deck

Definition 3.3.25

An **accessible deck** is one from which audience members with different vision, hearing, or color perception, and those using screen readers, can take in the content fully. **Alt text** is a short description that tells a screen reader what a visual is about.

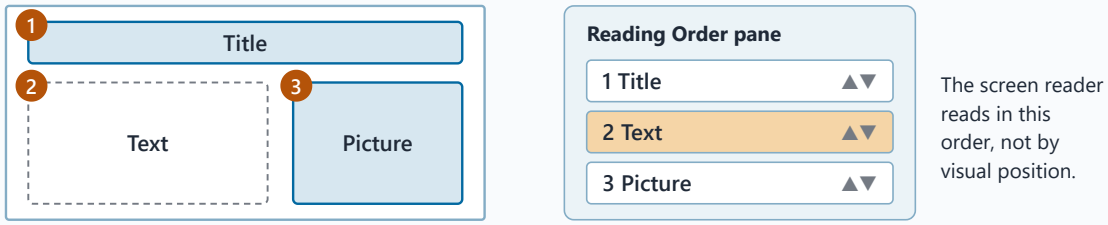


Figure 3.3.32: Reading order. The order in which a screen reader will read the Title, Text, and Picture on the slide is edited in the Reading Order pane. A screen reader reads objects not by their visual position but by the order in this list.

Six points are checked in an accessible deck:

- **Reading order:** A screen reader reads objects in the order they were added; this order may differ from the visual order (Figure 3.3.32). The order is corrected in the Reading Order pane.
- **Alt text:** Every meaningful visual is briefly described. Not “Picture” but “A column chart showing that enrollments rose 38% from 2023 to 2024”. Visuals that are purely decorative are marked “decorative”.
- **Title:** Every slide has a unique and descriptive title. An audience member who uses a screen reader navigates among slides by looking at the titles.

- **Contrast:** There is enough separation between text and background.
- **Color:** Information is not given by color difference alone; a label or symbol accompanies the color.
- **Captions:** Captions are used for video, audio, or live narration when appropriate.

Contrast is a measurable quantity. The Web Content Accessibility Guidelines (WCAG) recommend that the ratio between text color and background color be at least 4.5 : 1 for normal-size text. The ratio is calculated from the relative luminance of the two colors: (luminance of the lighter color + 0.05) ÷ (luminance of the darker color + 0.05). Luminance is 1 for white and 0 for black.

Example 3.3.20

Two text colors are tried on a white background: black and a medium gray with RGB values (119, 119, 119). Are the contrast ratios sufficient?

1. White on black: (1 + 0.05) ÷ (0 + 0.05) = 1.05 ÷ 0.05 = 21. The ratio is 21 : 1, the highest possible value.
2. For the gray, each channel is first brought to the 0–1 range: 119 ÷ 255 ≈ 0.467. Because screen colors are not linear, this value is converted to a luminance of $((0.467 + 0.055) \div 1.055)^{2.4} \approx 0.184$. Since the three channels are equal, the gray's relative luminance is also 0.184.
3. Ratio: (1 + 0.05) ÷ (0.184 + 0.05) ≈ 1.05 ÷ 0.234 ≈ 4.48. The ratio falls just below 4.5; according to the guideline this gray is insufficient for normal text.

A shade darker gray (118, 118, 118) gives about 4.54 : 1 and passes the limit. Because projectors show colors fainter than a screen, colors clearly darker than the borderline ones are chosen for a talk.

Common mistake

Showing the successful group in green and the unsuccessful group in red in a chart and making no other distinction. An audience member with red-green color blindness cannot tell the two groups apart, and a screen reader does not read color. A label (“successful”, “unsuccessful”) or a different pattern is added to the color.

3.3.9.2 Checkers before sharing

PowerPoint offers four checkers that scan the deck before sharing (Table 3.3.26).

Tool	Where	What it finds
Check Accessibility	Review	Images without alt text, untitled slides, low contrast, mixed reading order

Tool	Where	What it finds
Inspect Document	File → Info → Check for Issues	Comments, personal information such as the author, speaker notes, objects left outside the slide
Check Compatibility	File → Info → Check for Issues	Features that will not work in older PowerPoint versions
Spelling	Review	Spelling errors according to the language setting

Table 3.3.26: Checkers used before sharing.

The checkers find the problem but do not fix it. The accessibility checker says, for example, “Picture 3 has no alt text” and explains why. Writing the alt text or correcting the reading order is the job of the deck's owner. Inspect Document, on the other hand, can remove what it finds; but removed notes and comments cannot be brought back. That is why Inspect Document is run on a copy of the deck.

Common mistake

Running Inspect Document on the original file and deleting the speaker notes. The notes are lost with this operation in a way that cannot be undone. First a copy to be shared is made with Save As, and the inspection is done on that copy.

3.3.9.3 Collaboration and review

Decks are often prepared by a team. If the deck is stored in a supported cloud location such as OneDrive or SharePoint, several people can work on the same file at the same time (Figure 3.3.33).

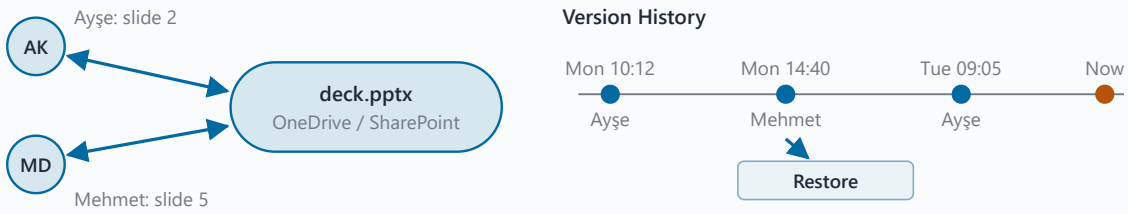


Figure 3.3.33: Co-authoring. Ayşe works on slide 2 and Mehmet on slide 5 of the same deck in the cloud at the same time. Version History lists earlier saves and the chosen version is brought back with Restore.

Tool	What it does
Comments	Feedback is written on a slide or an object; comments can be replied to and marked as resolved
Co-authoring	Simultaneous editing of a file in a cloud location; who is on which slide is visible
Version History	Access to earlier saved versions and restoring
Compare	Shows the differences between two separate copies of the deck
Spelling	Shows possible spelling errors according to the language setting

Tool	What it does
Final check	Comments, hidden slides, and personal information are reviewed before sharing

Table 3.3.27: Collaboration and review tools.

Common mistake

Team members circulating copies of the deck by e-mail and each making separate changes in theirs. In the end different files such as “deck_final_v3_reallyfinal.pptx” appear, and merging the changes takes hours. Co-authoring on a single cloud file prevents this problem from the start.

3.3.9.4 Recording and export

A deck is not only narrated live; it can be recorded and shared as a video, or distributed as a fixed document.

- **Record the talk:** Slide times, narration, camera video, and pointer movements are recorded slide by slide.
- **Export a video:** File → Export → Create a Video converts the recorded narration and timings into a watchable video file (.mp4). A default duration is used for slides without timings.
- **PDF:** Gives a fixed-layout share or print; it contains no animations or transitions.
- **Handouts:** Several slides are printed on one page; they can be distributed to the audience.
- **Packaging:** When the deck is carried to another computer, linked media and fonts are also checked.
- **File size:** The file is made smaller by compressing images and videos appropriately.

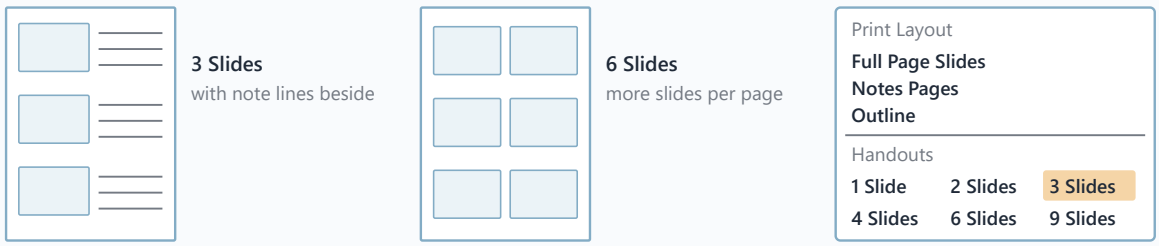


Figure 3.3.34: Handouts output. In the 3 Slides layout there are note lines beside each slide; in the 6 Slides layout more slides fit on the page. On the right, the print layouts in the Print settings are listed.

Common mistake

Adding a video as a link and copying the deck file alone to another computer. A linked video is not inside the deck file; it cannot be found on the new computer and does not play in the

show. The video is embedded in the deck or packaged together with the deck's media.

3.3.9.5 Print options

On the File → Print screen, the menu that says “Full Page Slides” determines what is printed on the page (Table 3.3.28).

Print layout	What is on the page?	For whom?
Full Page Slides	One slide on each page	The slides themselves
Notes Pages	Slide on top, speaker note below	The speaker
Outline	Only title and bullet text	The person reviewing the text
Handouts	1, 2, 3, 4, 6, or 9 slides per page; with note lines beside them in the 3-slide layout	Audience

Table 3.3.28: Print layouts.

Example 3.3.21

A 30-slide deck is to be printed. How many pages does it take in different layouts?

Layout	Calculation	Pages	Double-sided sheets
Full page	30 ÷ 1	30	15
3-slide handout	30 ÷ 3	10	5
6-slide handout	30 ÷ 6	5	3
9-slide handout	30 ÷ 9 ≈ 3.3 → 4	4	2

Table 3.3.29: The number of pages and sheets for the 30-slide deck.

1. The page count is the slide count divided by the slides per page, rounded up. 30 ÷ 9 ≈ 3.3 comes out; because the last page is printed even if only partly filled, 4 pages are needed.
2. In double-sided printing two pages fall on each sheet; the page count is divided by two and rounded up. In the 6-slide layout 5 ÷ 2 = 2.5 → 3 sheets.

The 9-slide layout is the most economical in paper. But when nine slides fit on an A4 page, each slide shrinks to roughly the size of a matchbox and small text cannot be read. For printouts to be distributed to the audience, the 3-slide or 6-slide layout is often chosen.

3.3.9.6 The final check and common mistakes

Before sharing or the day of the talk, the deck is reviewed one last time under six headings (Table 3.3.30).

Title	What to check
Content	Purpose, accuracy, sources, and narrative flow
Design	Title, font size, contrast, alignment, and spacing
Technical	Fonts, media, links, and animations
Accessibility	Title, reading order, alt text, and captions
Delivery	Notes, timing, the show, and screen settings
Sharing	File name, format, hidden slides, and personal information

Table 3.3.30: The final checklist.

The most common mistakes in talks and their fixes are collected in Table 3.3.31.

Common mistake	Consequence	Fix
Writing the whole speech on the slide	The audience reads and does not listen	Main point on the slide, detail in the notes
A different font and color on every slide	A disorderly, careless look	Theme and Slide Master
A different animation on every object	Attention shifts from the content to the motion	Few, purposeful effects
Small text and faint colors	The back rows cannot read it	Large point size, high contrast
Changing the slide size at the very end	Objects scale and the layout breaks	16 : 9 or 4 : 3 is chosen at the start
Copying a file with a linked video alone	The video does not open in the show	Embedded media or packaging
Giving information by color alone	An audience member who cannot tell colors apart misses it	Color + label or symbol

Table 3.3.31: Common talk mistakes.

3.3.10 Summary

- A **deck** is the file, a **talk** is the delivery. A slide supports the talk and does not replace it. The workspace consists of the ribbon, the slide pane, the slide area, the notes pane, and the status bar; **contextual tabs** appear while an object is selected.
- Views do not change the content: Normal is for editing, **Outline View** for building text, **Slide Sorter** for seeing and ordering the flow, and **Slide Master** for the common design.
- The plan starts with purpose, audience, and the **main message**. A content slide carries about 1–1.5 minutes; the question-and-answer time is written into the plan from the start.
- In the outline every top-level line is a slide; Tab lowers the level and Shift + Tab raises it.
- .pptx is editable, .ppsx is a direct show, .potx a template, .pdf fixed and animation-free, .mp4 a video. Save As creates a new file.

- The **layout** determines the type and position of placeholders. A hidden slide stays in the file and is skipped in the show. **Sections** group slides and move them group by group; they are not visible in the show.
- The **theme** chooses color and font, and the **Slide Master** carries their placement to all layouts and slides. Manual formatting locks only the property that was changed and takes precedence over the slide master; Reset removes this lock.
- In visual hierarchy the most important element is the largest and the source the faintest; the title states the result. Slide text is kept short and detail is moved to the notes.
- The ratio is kept when a picture is cropped. Pixels beyond what the screen can show only enlarge the file; Compress Pictures reduces to the target ppi and can delete cropped areas.
- The chart is chosen by the message: columns for comparison, line for time, pie for part-to-whole with few categories, scatter for relationships. An **embedded** chart travels with the file, a **linked** chart is updated from Excel.
- **Align** brings edges to a common line; **Distribute** leaves the two outer objects in place and equalizes the space: gap = (total span − sum of widths) ÷ (number of objects − 1).
- A **transition** works between slides, an **animation** on objects within a slide. The animation types are entrance, emphasis, exit, and motion path.
- Start = reference point + delay, end = start + duration. On Click takes the click, With Previous the start of the previous effect, and After Previous the end of the previous effects as the reference point. The numbers in the pane give the click count.
- Speaker notes are not visible to the audience; **Presenter View** shows the speaker the notes, the next slide, and the timer. F5 starts from the beginning, Shift + F5 from the current slide; B/W blacken or whiten the screen.
- The slide size is chosen at the start; in going from 16 : 9 to 4 : 3 the content shrinks by about 0.75. Fonts are embedded in the file.
- **Rehearse Timings** records slide times; a kiosk show stays in a loop, and the slide at moment t is found from the remainder of dividing by the lap time.
- Accessibility needs a title, reading order, alt text, enough contrast (at least 4.5 : 1 for normal text), and a cue other than color. The checkers find the problem; the user does the fixing; Inspect Document is run on a copy.
- In printing, the page count is the slide count divided by slides per page, rounded up; in double-sided printing the sheet count is half of that, rounded up.

Calculation	Formula	Example
Time per slide	talk time ÷ number of content slides	10 min ÷ 9 ≈ 1.1 min
Raw picture size	width × height × 3 bytes	4000 × 3000 → 36 MB
Target pixels	slide size (inches) × ppi	13.333 × 150 ≈ 2000
Distribute gap	(span between outer edges − widths) ÷ (n − 1)	(520 − 330) ÷ 2 = 95
Animation time	reference point + delay; + duration	1 + 0.25 = 1.25; 1.25 + 0.5 = 1.75 s

Calculation	Formula	Example
Slide in a loop	remainder of t divided by the lap time	46 − 40 = 6 → Slide 1
Contrast ratio	$(L_{light} + 0.05) \div (L_{dark} + 0.05)$	1.05 ÷ 0.234 ≈ 4.48
Print pages	[slides ÷ slides per page]	[30 ÷ 9] = 4

Table 3.3.32: The main calculations used in the notes.

3.3.11 Key terms

slide presentation file (deck) talk ribbon tab slide pane / notes pane outline Slide Sorter template placeholder slide layout hidden slide Summary / Section / Slide Zoom hyperlink theme Slide Master contrast pixel resolution compress (media) SmartArt linked / embedded object align / distribute group transition animation Animation Pane Animation Painter Designer Morph Copilot presenter Presenter View slide show slide size header / footer kiosk accessibility accessibility checker alt text

3.3.12 Test Yourself

1. In Outline View the following six lines are typed in order: Budget, Income, Scholarship, Rent, Expense, Conclusion. Then Tab is pressed on the “Scholarship” and “Rent” lines, and afterwards Shift + Tab on the “Rent” line. How many slides are there at each stage, and what are the bullets of the “Income” slide in the final state?

   Answer

   All six lines are at title level: 6 slides. When Tab is pressed on Scholarship and Rent, both become bullets of Income: Budget, Income, Expense, Conclusion, that is, 4 slides. Shift + Tab on Rent raises the level again and Rent becomes a separate slide: Budget, Income, Rent, Expense, Conclusion, that is, 5 slides. The only bullet of the Income slide is Scholarship.

2. 6 minutes of a 20-minute session are set aside for questions. If a content slide carries 1–1.5 minutes, how many content slides should be planned? How many seconds per slide does a student get who prepares 35 slides for the same time?

   Answer

   14 minutes remain for the talk: 20 − 6. 14 ÷ 1.5 ≈ 9.3 and 14 ÷ 1 = 14, so about 10–14 content slides are suitable. With 35 slides each slide gets 14 × 60 ÷ 35 = 24 seconds; that is not enough time to explain a chart.

3. On the slide master the title size is 40 pt and the color navy. Slide 2's title had earlier been given only a manual size of 28 pt. On the slide master the title size is now changed to 44 pt and the color to maroon. After Close Master View, how do the titles of Slide 1 and Slide 2 look? What happens if Reset is applied to Slide 2?

   Answer

   Slide 1: 44 pt, maroon; both rules come from the slide master. Slide 2: the size stays 28 pt because it was given by hand, and the color becomes maroon because it was not changed by hand. Reset deletes the manually given size; Slide 2's title also becomes 44 pt and maroon.

4. A 6000 × 4000-pixel (3 : 2) photo will be shown full screen on a 16 : 9 slide on a 1920 × 1080-pixel projector. Counting 3 bytes per pixel, how many times the amount the screen can display is the photo's raw size? If it is cropped to 16 : 9 keeping the width, what percentage of the pixels remains?

   Answer

   The photo is 6000 × 4000 × 3 = 72,000,000 bytes (72 MB), and the screen image 1920 × 1080 × 3 = 6,220,800 bytes. The ratio is 72,000,000 ÷ 6,220,800 ≈ 11.6. On cropping, the height becomes 6000 × 9 ÷ 16 = 3375 pixels; 4000 − 3375 = 625 rows are discarded. The remaining ratio is 3375 ÷ 4000 ≈ 84.4%.

5. On a slide extending horizontally from 0 to 960 there are three shapes: A (x = 40, width 120), B (x = 300, width 100), and C (x = 700, width 200). All three are selected and Distribute Horizontally is applied; then they are grouped and centered horizontally. What are the shapes' x values after each step?

   Answer

   The outer edges are 40 and 700 + 200 = 900; the span is 860. The sum of the widths is 120 + 100 + 200 = 420, the empty space 860 − 420 = 440, each gap 220. After Distribute: A = 40, B = 40 + 120 + 220 = 380, C = 700. The group is 860 wide; when centered its left edge becomes (960 − 860) ÷ 2 = 50, that is, the group shifts 10 units to the right: A = 50, B = 390, C = 710.

6. In the Animation Pane there are these effects in order: Title (On Click, duration 1 s), Subtitle (With Previous, delay 0.5 s, duration 1 s), Bullet (After Previous, duration 0.5 s), Logo (After Previous, delay 1 s, duration 2 s), Chart (On Click, duration 1.5 s). The presenter clicks at t = 0 and t = 8.

When does each effect play? Which numbers appear in the pane?

Answer

Title 0–1 s. Subtitle 0.5 s after the Title's start: 0.5–1.5 s. Bullet starts at the latest end of the previous effects (1.5): 1.5–2 s. Logo, 1 s after the end of Bullet: 3–5 s. Chart with the second click: 8–9.5 s. In the pane only 1 appears next to Title and 2 next to Chart; two clicks are enough for the slide's animations.

7. Slide 1 has no animation. Slide 2 has been given a Fade transition. Bullet 1 and Bullet 3 start On Click and Bullet 2 With Previous; Bullet 3 has also been given an emphasis that starts After Previous. In a show starting from Slide 1, how many clicks are needed to see Slide 3?

Answer

Four clicks. Click 1 moves to Slide 2 with Fade. Click 2 brings Bullet 1 and Bullet 2 together. Click 3 brings Bullet 3; the emphasis then plays by itself. Click 4 moves to Slide 3.

8. In a three-slide kiosk show the timings are 15, 20, and 25 s. What slide is on the screen 100 s and 200 s after the show starts, and how many seconds have passed in that slide?

Answer

The lap time is 15 + 20 + 25 = 60 s; the intervals are Slide 1 0–15, Slide 2 15–35, Slide 3 35–60. t = 100: 100 − 60 = 40, Slide 3's interval; 40 − 35 = 5 s have passed in the slide. t = 200: 200 − 3 × 60 = 20, Slide 2's interval; 20 − 15 = 5 s have passed in the slide.

9. A 45-slide deck will be printed double-sided as a 6-slide handout. How many pages and how many sheets are needed? What is the page count in the 4-slide layout?

Answer

45 ÷ 6 = 7.5 → 8 pages; in double-sided printing 8 ÷ 2 = 4 sheets. In the 4-slide layout 45 ÷ 4 = 11.25 → 12 pages.

10. In an eight-slide deck slide 7 is hidden. A custom show named “Jury” consists of slides 1, 2, 5, and 8. How many slides are shown with F5? Which slides does the “Jury” show present? Is it safe to share the file because of the hidden slide?

Answer

With F5 seven slides are shown in the order 1, 2, 3, 4, 5, 6, 8; slide 7 is skipped. The “Jury” show presents only 1, 2, 5, and 8. Because the hidden slide stays in the file, anyone who receives the file can see it; if there is sensitive content, it is deleted before sharing.

3.3.13 Further reading

- Microsoft Support, *PowerPoint help and learning* (support.microsoft.com/powerpoint): the official help pages that explain slide layouts, the Slide Master, transitions, animation, Presenter View, and export with current menu names and short videos.
- Microsoft Support, *Make your PowerPoint presentations accessible to people with disabilities*: gives step by step the use of alt text, reading order, slide titles, and the Accessibility Checker.
- W3C, *Web Content Accessibility Guidelines (WCAG) 2.2*: the international accessibility guideline on which the contrast ratio and the rule about cues other than color are based; the definition of the relative-luminance formula is here.

3.3.14 References

1. Microsoft Support, PowerPoint help and learning (support.microsoft.com/powerpoint).
2. Microsoft Support, Add, change or remove transitions between slides.
3. Microsoft Support, Give a presentation in PowerPoint (Presenter View).
4. Microsoft Support, Benefits of embedding custom fonts.
5. Microsoft Support, Frequently asked questions about Copilot in Microsoft 365 subscriptions.
6. W3C, Web Content Accessibility Guidelines (WCAG) 2.2, 2023.

PART II

From Algorithm to Program

Chapters 4–5

PART II · CHAPTER 4

Algorithms and Introduction to Programming

Programming means turning the steps of a solution into instructions a computer can carry out. The Algorithms deck covers problem definition, pseudocode and the qualities of a good algorithm, works through examples such as factorial, Fibonacci and prime numbers, and traces linear and binary search as well as selection, bubble, insertion and merge sort. The Flowcharts deck shows sequential, conditional and repeated flow, along with counters and accumulators. Introduction to Programming presents programming languages, compilers and interpreters, programming paradigms, the software development process and debugging.

NOTES IN THIS CHAPTER

4.1 Algorithms

4.2 Flowcharts

4.3 Introduction to Programming

PART II · CHAPTER 4.1

Algorithms

Whether a program will work is often settled before the first line of code is written. If the problem has been defined precisely, the solution worked out step by step, and a few small examples tried by hand, coding becomes a fairly mechanical translation. This chapter starts with defining a problem, then covers the properties of a good algorithm, the algorithm design cycle, and pseudocode. Five small algorithms are then traced line by line: ordering two numbers, factorial, the Fibonacci sequence, palindrome checking, and primality testing. We check an algorithm's correctness with a trace table and test cases, and measure its efficiency by counting operations. The last two sections turn to two of the most basic tasks in computer science: linear and binary search, and selection, bubble, insertion, and merge sort. Each method is run step by step on an example array, and every comparison it makes is counted.

LEARNING OUTCOMES

Defining
Identify a problem's goal, input, output, constraints, check example, and error case.

Designing
Break a solution into clear, ordered, executable, finite steps, and write the algorithm in pseudocode.

Verifying
Run an algorithm by hand with a trace table, and choose test cases made up of normal, boundary, invalid, and large inputs.

Comparing
Evaluate linear and binary search, and selection, bubble, insertion, and merge sort, in terms of number of comparisons, time complexity, and space complexity.

4.1.1 Problem solving

A beginner who reads an assignment usually wants to sit down at the computer and start typing code right away. For short assignments this sometimes works. Once the problem grows a little, the code swells with trial and error, fixing one bug introduces another, and it becomes unclear why the program works or why it does not. Experienced programmers do the opposite: they first understand the problem, then work out the solution on paper, and write the code last. **Problem solving** is the name of this disciplined approach.

4.1.1.1 Algorithm first, code second

Code is the expression of a solution idea in a particular programming language. The same idea can be written in Java, in Python, or in C; only the syntax changes. The idea itself, the sequence of steps that solves the problem, is independent of the language. This sequence of steps is called an **algorithm**. Code written before the algorithm is clear causes two troubles. First, the programmer decides what to do while typing and so makes needless experiments. Second, some cases are never considered, which leaves hidden bugs: the program works on the sample input and gives a wrong result on another.

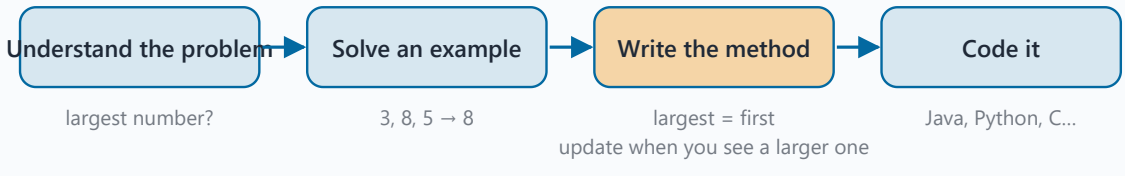


Figure 4.1.1: The right order for solving a problem: understand the problem, solve a small example by hand, write the method, then code it. The example problem is finding the largest number in a list. The orange step (the algorithm) is independent of the language; the same method can be coded in any language.

Figure 4.1.1 shows this order on the problem “find the largest number.” The first step is **understanding the problem**: can the list be empty, can the numbers be negative, can the same number occur more than once? In the second step, **a small example is solved by hand**. In the list 3, 8, 5 the largest number is 8. Without noticing, a person follows a method while finding it: look at the numbers from left to right and remember the largest value seen so far. In the third step this method is **written down** explicitly: “The largest value starts as the first element of the list; if the next element is larger, update the largest value.” The fourth step is **coding**; the method is now settled, and all that remains is to apply the syntax rules of the chosen language.

The second step is often skipped, yet it is the most instructive. The example solved by hand gives a draft of the algorithm. Every operation performed while solving the example becomes a step of the algorithm, and every piece of information kept in mind becomes a variable. In the largest-number

example the information kept in mind is “the largest value so far”; in the program this becomes a variable called `largest`.

4.1.1.2 Problem definition

To understand a problem is to write it down in a way that answers a few questions. Together, the answers to these questions form the **problem definition**.

Definition 4.1.1

A **problem definition** states six elements of the problem to be solved: the **goal** (which problem is to be solved), the **input** (what information is given to the program), the **output** (what result is produced), the **constraint** (the limits and validity conditions of the values), the **example** (which small case will verify the expected result), and the **error case** (what to do on invalid or missing input).

Each of these six elements settles in advance a decision that would otherwise have to be made while writing code. The goal determines when the program counts as “finished.” The input and output say how the program talks to the outside world. The constraint limits the values for which the program must work correctly. The example prepares the first check to run once the program is written. The error case is the element most often forgotten: it is what makes the program give a meaningful message, instead of crashing, when the user enters an unexpected value.

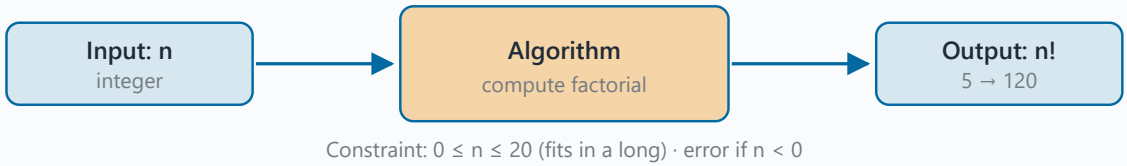


Figure 4.1.2: Summary of the problem definition for computing a factorial. The input is an integer n, the output is n!; the constraint is $0 \le n \le 20$ (the result fits in the 64-bit long type), and an error is reported if $n < 0$.

Example 4.1.1

How is a complete problem definition written for the request “compute the factorial of a number”?

The factorial is the product of the integers from 1 to n: $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$. By definition, $0! = 1$. The six elements of the problem definition are filled in as follows:

Element	Answer for the factorial problem
Goal	Compute the factorial of the given number n
Input	An integer n
Output	The value of n!
Constraint	0 ≤ n ≤ 20 (the result will be held in the 64-bit long type)
Example	120 for n = 5; 1 for n = 0
Error case	If n < 0 or n > 20, print “error”

The limit of 20 in the constraint row is not arbitrary. Java's `long` type holds at most 9,223,372,036,854,775,807. *20! = 2,432,902,008,176,640,000* stays below this limit, while *21! = 51,090,942,171,709,440,000* goes above it.

Common mistake

Writing only the input and output in the problem definition and skipping the constraint and the error case. The sentence “compute the factorial of n” does not say what happens when n = −3 is entered. If this question is not answered at design time, the program either gives a meaningless result or enters a loop that never stops. The constraint and the error case are at least as important as the input and output.

4.1.2 Algorithm

4.1.2.1 What is an algorithm?

Everyday life is full of algorithms. A recipe, the assembly instructions of a piece of furniture, and the steps for withdrawing money from an ATM can all be called algorithms. In each of them there are steps to be carried out in order to reach a particular result. In computer science the notion of an algorithm is defined more precisely, because the one carrying out the steps is not a person but a machine that cannot interpret anything on its own.

Definition 4.1.2

An **algorithm** is a finite sequence of well-defined steps for solving a particular problem. An algorithm takes zero or more inputs and produces at least one output.

The phrase “zero or more inputs” in the definition may look odd at first. Some algorithms take no input at all: the task “print the numbers from 1 to 100” asks for no information from outside. An algorithm with no output, on the other hand, is useless; a computation whose result is reflected

nowhere is no different from a computation that was never done. The output may be a number printed on the screen, but it may equally be data saved to a file or a signal that starts a motor.

4.1.2.2 Properties of a good algorithm

Not every sequence of steps is a good algorithm. The sentence “arrange the numbers appropriately” looks like a step, but the word “appropriately” does not say what to do. For an algorithm to run reliably on a computer, its steps must have certain properties.

Property	Meaning	What happens when it fails
Definiteness	Every step is clear and has a single meaning	Different people carry out a step like “enlarge it a bit” differently
Ordering	The order in which steps are performed is fixed	If the order of assignments in a swap changes, a value is lost
Feasibility	Every step can actually be carried out	The step “list all prime numbers” can never be completed
Finiteness	The algorithm stops after a finite number of steps	The loop condition never becomes false, so the program never stops
Correctness	Produces the right result for every valid input	The program gives a wrong result on some inputs
Efficiency	Resource use stays acceptable as the input grows	A program that is fast on small data runs for hours on large data

Table 4.1.1: Properties of a good algorithm.

Table 4.1.1 lists six properties and the symptom seen when each one fails. Of these, **definiteness** and **finiteness** are the most often overlooked. A small example shows how subtle the condition for finiteness can be.

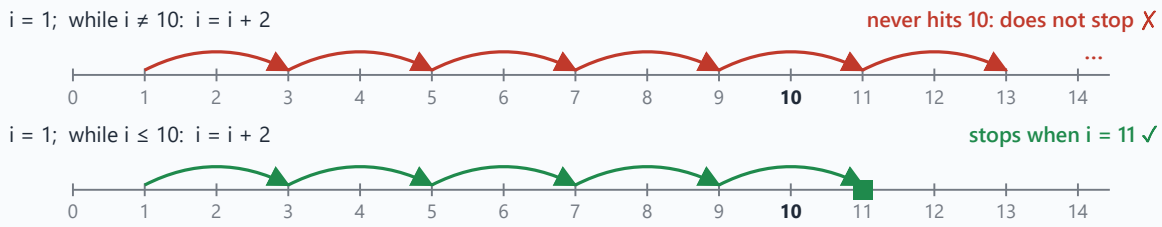


Figure 4.1.3: Finiteness. In the upper loop, i starts at 1 and grows by two: 1, 3, 5, … The value is never exactly 10, so the condition i ≠ 10 stays true and the loop never stops. In the lower loop the condition is i ≤ 10; when i = 11 the condition becomes false and the loop stops.

Figure 4.1.3 compares two loops. In both, the counter starts at 1 and increases by 2 on every iteration. The upper loop says “keep going as long as i is not equal to 10.” The counter runs 1, 3, 5, 7, 9, 11, 13 … and skips over 10. Since the condition never becomes false, the loop goes on forever. The lower loop

says “keep going as long as i is less than or equal to 10.” When the counter reaches 11, *11 ≤ 10* is false and the loop ends. The difference between the two conditions is a single symbol, but one of them is an algorithm and the other is not.

Common mistake

Using an equality condition (= or ≠) to end a loop. If the counter never lands exactly on that value, the loop never stops. When a counter moves toward a limit, a “less than or equal” (≤) or “less than” (<) condition is safer; the moment the counter passes the limit, the loop is guaranteed to end.

Correctness and **efficiency** are separate properties. A correct algorithm can be slow, and a fast algorithm can give a wrong result on some inputs. Correctness comes first; a fast but wrong result has no value. Efficiency gains importance as the input grows.

4.1.2.3 The algorithm design cycle

A good algorithm rarely appears on the first attempt. Design is a cycle repeated several times. On each pass the algorithm becomes a little clearer and overlooked cases are caught.

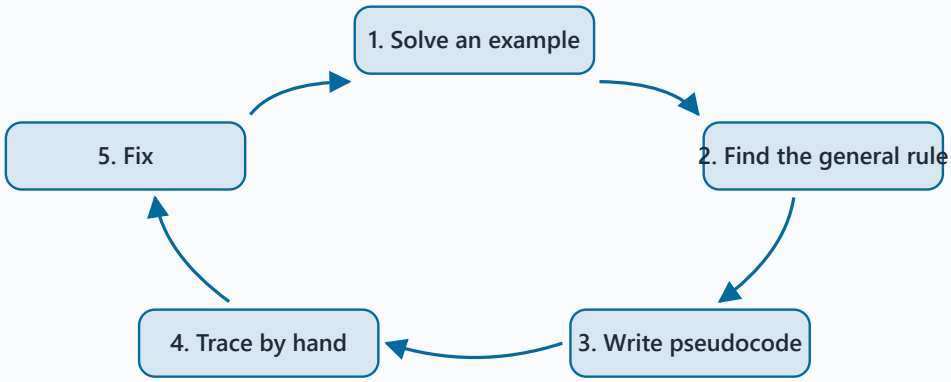


Figure 4.1.4: The algorithm design cycle: solve an example, find the general rule, write pseudocode, trace by hand, fix. If the hand trace reveals an error, it is corrected and the cycle runs again; when no errors remain, we move on to coding.

The five steps in Figure 4.1.4 work as follows:

1. **Solve an example.** A small instance of the problem is solved by hand. For the factorial, we compute *4! = 1 × 2 × 3 × 4 = 24*.

2. **Find the general rule.** We notice the repetition in the hand computation: at every step, the product so far is multiplied by the next number. The rule is written as "start the product at 1 and multiply it by every number from 2 to n."
3. **Write pseudocode.** The rule is turned into clear steps that are close to a programming language but not tied to one.
4. **Trace by hand.** The pseudocode is run line by line on a few inputs, like a computer would, and the value of each variable is written in a table.
5. **Fix.** If the trace shows a difference between the expected and the actual result, the error is located and the pseudocode is corrected. The cycle then runs again: the corrected pseudocode is traced once more.

The cycle ends when the hand trace no longer reveals errors, and coding begins. If several solutions have been found for the same problem, they are compared at this stage for correctness, readability, and efficiency. Readability should not be underestimated: the person who reads an algorithm most is the programmer who has to modify it a few months later.

4.1.2.4 Pseudocode

There are several ways to write down an algorithm. Steps written in everyday language are easy to understand but open to ambiguity. Steps written in a programming language are precise but full of syntax rules; semicolons and parentheses get in the way of thinking. **Pseudocode** sits between these two extremes.

> **Definition 4.1.3**
>
> **Pseudocode** is a way of writing the logic of an algorithm in short, consistent statements close to natural language, without tying it to any particular programming language.

Pseudocode has no single universally accepted syntax. What matters is that within one algorithm the same statement always has the same meaning. The statements used in this chapter are: `read` takes an input, `print` produces an output, `=` assigns the value on the right to the variable on the left, `if … then` checks a condition, `while …` repeats the lines below it as long as the condition is true, and `for … to …` repeats a fixed number of times. **Indentation**, the whitespace at the start of a line, shows which lines a condition or loop covers.

The simplest example is an algorithm that prints the sum of two numbers:

```
START
  read A
  read B
  SUM = A + B
  print SUM
END
```

The same algorithm can also be drawn as a **flowchart**. Figure 4.1.5 shows the counterpart of each pseudocode line in the diagram: start and end are ovals, input and output are parallelograms, and a computation is a rectangle. Arrows show the order of the steps.

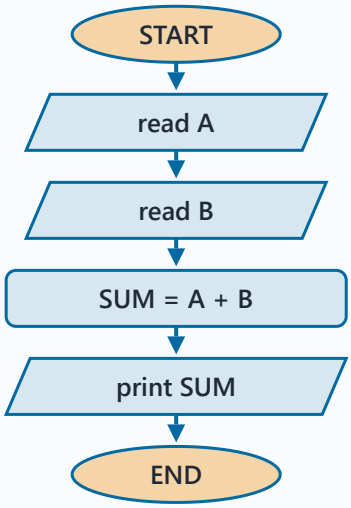


Figure 4.1.5: Flowchart of the addition algorithm: start, read A, read B, SUM = A + B, print SUM, end.

The goal of pseudocode is not to follow syntax rules but to make the algorithm understandable. That is why a line such as “swap arr[i] and arr[j]” can be written freely in pseudocode. In Java the same job takes three assignments, but from the point of view of the algorithm's logic it is a single step. Once the pseudocode is ready, translating it into code is often a line-by-line task: an `if … then` line becomes `if`, and a `while …` line becomes `while`.

Common mistake

Writing pseudocode that is either too vague or too detailed. The line “process the list and find the result” says nothing. At the other extreme, pseudocode that puts a semicolon at the end of every line and declares every variable type is really Java code and loses the convenience of pseudocode. The yardstick is this: someone reading the pseudocode should be able to write the algorithm in whatever language they know.

4.1.3 Example algorithms

The best way to learn to build algorithms is to solve small, well-known problems. The five examples in this section get progressively harder. The first contains only one condition and three assignments. The factorial and Fibonacci examples bring in loops and the idea of accumulation. In the palindrome and prime examples, the loop ends early as soon as the result is known. In each example the pseudocode is given first, and then the algorithm is traced step by step on one input.

4.1.3.1 Ordering two numbers and swapping

The problem: read two numbers and print the smaller one first. If a = 7, b = 3 is entered, the output must be “3 7”; for a = 2, b = 9, it is “2 9”. The idea is simple. If a is greater than b, the values of the two variables are exchanged; otherwise they are already in the right order. Exchanging the values of two variables is called a **swap**.

```
read a, b
if a > b then
    temp = a
    a = b
    b = temp
print a, b
```

Line 2 of the algorithm checks the condition. If it is true, the three indented lines run; if it is false, they are skipped. When the values are equal (a = b), the condition is false and no swap happens, which is fine because none is needed. The last line runs in both cases and prints output satisfying a ≤ b.

Why a swap needs three assignments is the most instructive part of the algorithm. A variable can hold only one value at a time. The moment a new value is written into a with `a = b`, its old value is erased. If the old value has not been copied somewhere beforehand, it can never be recovered. That is why the value of a is first copied into a **temporary variable**.

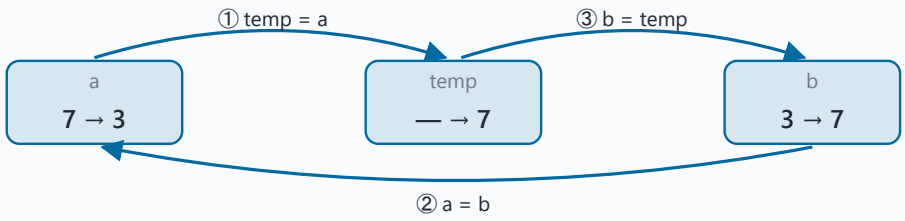


Figure 4.1.6: The three assignments of a swap. ① temp stores a's value 7. ② a takes the value 3 from b; a's old value now lives only in temp. ③ b takes the value 7 from temp.

Figure 4.1.6 shows the three assignments with arrows. The assignments form a cycle: from a to temp, from b to a, from temp to b. The step-by-step run of the algorithm for a = 7, b = 3 is given in Table 4.1.2.

Step	Line executed	a	b	temp
1	`read a, b`	7	3	—
2	`if 7 > 3 then`	7	3	—
3	`temp = a`	7	3	7
4	`a = b`	3	3	7
5	`b = temp`	3	7	7
6	`print a, b → 3 7`	3	7	7

Table 4.1.2: Trace table of the two-number ordering algorithm for a = 7, b = 3. Each row shows the values after that line has run; "—" means the variable has not yet received a value.

In row 4 of the table, a and b are both 3. This is the most fragile moment of the algorithm: the value 7 lives only in temp. In row 5 the 7 in temp is copied to b and the swap is complete. Had a = 2, b = 9 been entered, the condition *2 > 9* in row 2 would be false, rows 3–5 would never run, and "2 9" would be printed directly.

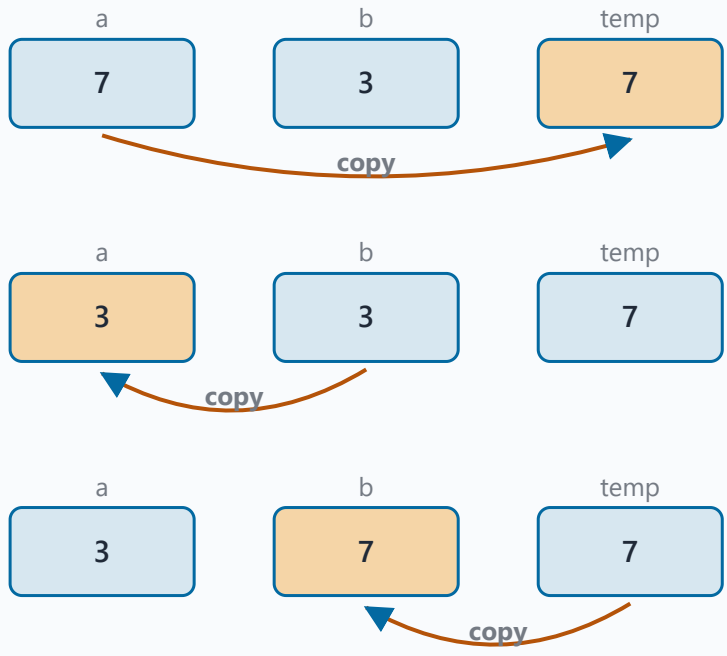


Figure 4.1.7: The three moments of the swap (in order): after temp = a (a = 7, b = 3, temp = 7); after a = b (a = 3, b = 3, temp = 7); after b = temp (a = 3, b = 7, temp = 7).

Common mistake

Writing the swap without a temporary variable, with two assignments `a = b` and `b = a`. For a = 7, b = 3, the first assignment makes a equal to 3 and erases the 7. The second assignment writes a's new value, 3, into b. The result is “3 3”; the 7 is lost. Assignment is a copying operation; it does not exchange the contents of two boxes at the same time.

The idea of ordering two numbers can be extended to more numbers. For three numbers, the two-number ordering step is applied three times: first a with b, then b with c, and finally a with b again.

Example 4.1.2

How does the algorithm below work for a = 8, b = 5, c = 1? Why are three comparisons enough?

```
read a, b, c
if a > b then swap a and b
if b > c then swap b and c
if a > b then swap a and b
print a, b, c
```

Step	Condition	Result	a	b	c
start	—	—	8	5	1
1st comparison	a > b	true → a and b are swapped	5	8	1
2nd comparison	b > c	true → b and c are swapped	5	1	8
3rd comparison	a > b	true → a and b are swapped	1	5	8

The output is “1 5 8”. Because the input was in reverse order, all three comparisons caused a swap. The first two comparisons carry the largest value to the far right, like a bubble: after step 1 the 8 moves to b, and after step 2 to c. Now c definitely holds the largest value. What remains is to order a and b, which the 3rd comparison does. That is why three comparisons are always enough for three numbers.

4.1.3.2 Factorial

The **factorial** of n is the product of the integers from 1 to n, written n!. *3! = 1 × 2 × 3 = 6* and *5! = 120*. By definition, 0! is 1. The problem definition of the factorial was written in 4.1.1.2; the algorithm follows that definition step by step.

```
read n
if n < 0 then print "error" and stop
result = 1
i = 2
while i ≤ n
    result = result × i
    i = i + 1
print result
```

The lines do the following:

1. `read n`: the number whose factorial is to be computed is read.
2. `if n < 0 then print "error" and stop`: the factorial of a negative number is not defined. This line handles the error case in the problem definition.
3. `result = 1`: the starting value of the product is 1. The variable builds up the product step by step.
4. `i = 2`: the counter starts at 2. Multiplying by 1 does not change the value, so there is no need to start at 1.
5. `while i ≤ n`: as long as i does not exceed n, the two lines below it are repeated.
6. `result = result × i`: the product so far is multiplied by i.
7. `i = i + 1`: the counter increases by one; then control returns to line 5 and the condition is checked again.
8. `print result`: when the loop ends, the result is printed.

A variable like the one in line 3, which builds up a result step by step over a loop, is called an **accumulator**. The accumulator's starting value is chosen according to the operation. For addition we choose 0, because adding 0 does not change the sum. For multiplication we choose 1, because multiplying by 1 does not change the product.

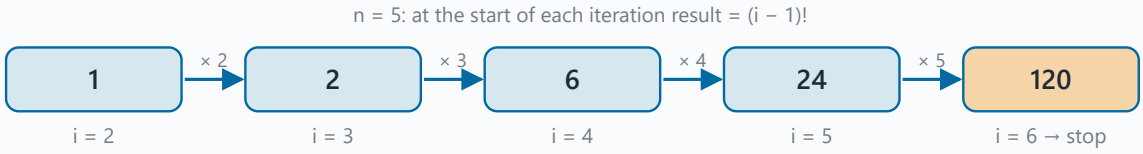


Figure 4.1.8: The factorial chain for n = 5. At the start of each iteration result = (i − 1)!: 1 at i = 2, 2 at i = 3, 6 at i = 4, 24 at i = 5. In the last iteration 24 × 5 = 120 is obtained; the loop stops when i = 6.

Figure 4.1.8 shows how the product grows. The full trace of the algorithm for n = 5 is given in Table 4.1.3.

State	Condition i ≤ n	Multiply	result	New i
start	—	—	1	2

State	Condition i ≤ n	Multiply	result	New i
Iteration 1	2 ≤ 5 → true	1 × 2 = 2	2	3
Iteration 2	3 ≤ 5 → true	2 × 3 = 6	6	4
Iteration 3	4 ≤ 5 → true	6 × 4 = 24	24	5
Iteration 4	5 ≤ 5 → true	24 × 5 = 120	120	6
exit	6 ≤ 5 → false	—	120	6

Table 4.1.3: Trace table of the factorial algorithm for n = 5. The "Condition" column is the condition i ≤ n checked at the start of the iteration.

The table shows that the loop runs 4 times: i takes the values 2, 3, 4, and 5. In general, for n ≥ 2 the loop runs $n - 1$ times. When i = 6, the condition $6 \leq 5$ is false and the loop ends; 120 is printed.

A pattern stands out in the result column of the table. At the start of every iteration, result equals the factorial of one less than i: 1 = 1! when i = 2, 2 = 2! when i = 3, 6 = 3! when i = 4. A statement like this, which stays true on every iteration of the loop, is called a **loop invariant**.

Definition 4.1.4

A **loop invariant** is a statement that is true at the start of every iteration of a loop. The loop invariant of the factorial algorithm is "at the start of each iteration, $result = (i - 1)!$."

The loop invariant shows why the algorithm is correct. When the loop starts for the first time, i = 2 and result = 1 = 1!; the statement holds. In one iteration, result is multiplied by i and i increases by one. If $result = (i - 1)!$ before the iteration, then after it $result = (i - 1)! \times i = i!$. Since the new i is one more than the old i, the statement holds again. The loop ends when $i = n + 1$; at that moment $result = n!$, which is exactly the desired result.

The invariant also explains how the algorithm behaves on small inputs. For n = 0 or n = 1, the condition $2 \leq n$ is false at the first check. The loop never runs and result stays 1; 0! and 1! are 1 anyway. The algorithm needs no separate line for these two cases.

Common mistake

Giving the accumulator the wrong starting value, or writing the loop bound one short. If `result = 0` is written, every product comes out 0 and the algorithm prints 0 for every n. If the condition is written `i < n` instead of `i ≤ n`, the last multiplication is skipped; for n = 5 the result is 24 instead of 120. The second mistake is called an **off-by-one error**. An off-by-one error does not show up for the inputs n = 0 and n = 1, because the loop does not run there anyway. That is why an algorithm should never be tried on a single input and then left.

The accumulator idea is used not only for the factorial but in every problem that needs a sum, a count, or an average. The following example applies the same pattern to addition.

Example 4.1.3

How is an algorithm written that finds the sum of the even numbers from 1 to n? How does it work for n = 10?

Since we are adding, the accumulator starts at 0. Even numbers start at 2 and grow by two; the counter likewise starts at 2 and is increased by 2 on every iteration. This way the odd numbers are never looked at.

```
read n
sum = 0
i = 2
while i ≤ n
    sum = sum + i
    i = i + 2
print sum
```

State	Condition i ≤ n	sum	New i
start	—	0	2
Iteration 1	2 ≤ 10 → true	2	4
Iteration 2	4 ≤ 10 → true	6	6
Iteration 3	6 ≤ 10 → true	12	8
Iteration 4	8 ≤ 10 → true	20	10
Iteration 5	10 ≤ 10 → true	30	12
exit	12 ≤ 10 → false	30	12

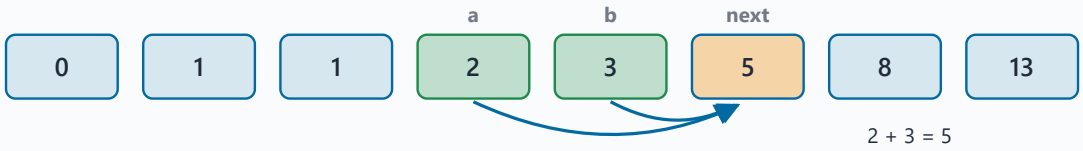
The result is *2 + 4 + 6 + 8 + 10 = 30*. The loop runs 5 times; when the counter reaches 12 the condition becomes false. For n = 0 or n = 1 the condition is false at the first check and 0 is printed; this is correct, since there is no positive even number less than or equal to 1. It matters that the loop condition uses ≤: for n = 10, 10 itself must be included in the sum.

4.1.3.3 The Fibonacci sequence

The Fibonacci sequence starts with 0 and 1; every following term is the sum of the two terms before it: 0, 1, 1, 2, 3, 5, 8, 13, 21, … The problem is to print the first n terms of the sequence. In the factorial it was enough to remember a single value on each iteration. Here, computing the new term requires keeping the two previous values together.

```
read n
a = 0, b = 1
for round = 0 to n-1
    print a
    next = a + b
    a = b
    b = next
```

a and b hold two consecutive terms of the sequence. On each iteration a is printed, then a and b are added to find the next term. Then the two variables advance by one term: a takes the value of b, and b takes the value of the new term. Together, a and b are like a two-cell window that slides one position to the right along the sequence on each iteration. This technique is called a **sliding window**.



Round 3: a (2) is printed, next = a + b; then a = 3, b = 5 and the window slides one to the right.

Figure 4.1.9: The sliding window on the Fibonacci sequence. In round 3, a = 2 is printed and next = a + b = 2 + 3 = 5 is computed; then a = 3, b = 5 and the window slides one to the right.

Figure 4.1.9 shows the state in the third round. All rounds of the algorithm for n = 8 are traced in Table 4.1.4.

Round	a printed	next = a + b	New a	New b
start	—	—	0	1
0	0	0 + 1 = 1	1	1
1	1	1 + 1 = 2	1	2
2	1	1 + 2 = 3	2	3
3	2	2 + 3 = 5	3	5
4	3	3 + 5 = 8	5	8
5	5	5 + 8 = 13	8	13
6	8	8 + 13 = 21	13	21
7	13	13 + 21 = 34	21	34

Table 4.1.4: Trace table of the Fibonacci algorithm for n = 8. Rounds are numbered from 0 to n − 1 = 7; “New a” and “New b” are the values at the end of the round.

The output is the sequence in the “a printed” column: 0 1 1 2 3 5 8 13. In the last round next = 34 is computed but never printed; the loop ends after 8 rounds. This small waste is the price paid for the simplicity of the algorithm. For n = 0 the algorithm prints nothing, and for n = 1 it prints only 0; both are the expected results.

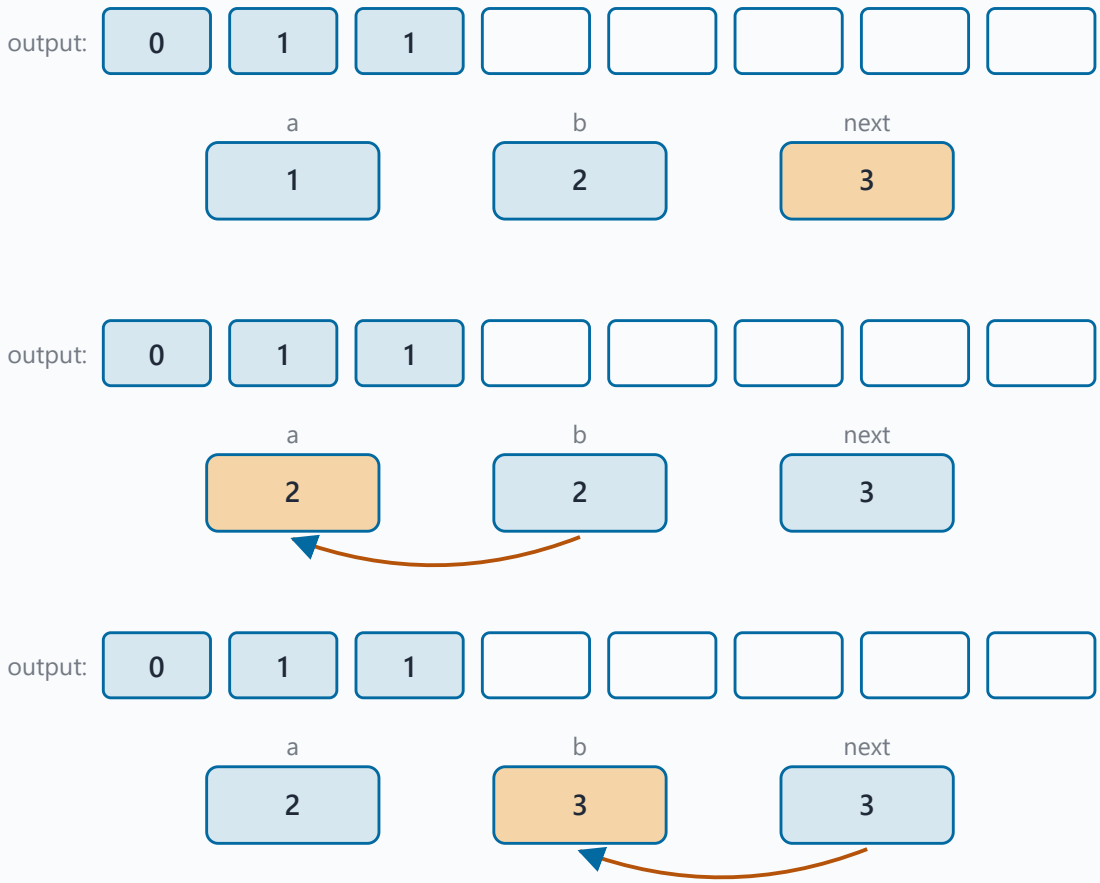


Figure 4.1.10: Fibonacci, the last three steps of round 2 (in order): next = 1 + 2 = 3 is computed; with a = b, a becomes 2; with b = next, b becomes 3. The top row shows the terms printed so far.

The order of the last two assignments raises a problem similar to the one in the swap. As in Figure 4.1.10, first `a = b` is executed, then `b = next`. If the order is reversed, the assignment `b = next` erases the old value of b. Then `a = b` writes the new term into a, not the old b. a and b become equal, and on every round the new term is computed as $a + a = 2a$. The sequence then doubles each time: 0, 1, 2, 4, 8.

Common mistake

Ignoring the order of interdependent assignments. Before a new value is written to a variable, every operation that needs its old value must be finished. In Fibonacci, the old b is needed for a; that is why `a = b` is written first. The temporary variable (here next) preserves this order

by storing the new value the moment it is computed.

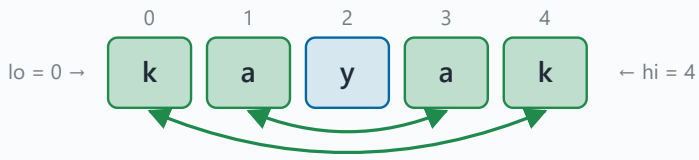
4.1.3.4 Palindromes

A word or sentence that reads the same backward as forward is called a **palindrome**. “kayak,” “ses,” and “kazak” are palindromes (they are Turkish words, for boat, voice, and sweater); “kayık” is not. The problem is to report, as true or false, whether a given **text** is a palindrome.

The first solution that comes to mind is to reverse the text and compare it with itself. This is correct, but it creates a copy of the text. A solution that does less work starts at the two ends of the text and moves toward the middle. The first letter is compared with the last, the second with the second from last. If a differing pair is found, the text is not a palindrome and false is returned at once. If no differing pair appears before the ends meet in the middle, the text is a palindrome.

```
read text
clean text by the rule
lo = 0, hi = length - 1
while lo < hi
    if the characters differ, return false
    lo = lo + 1, hi = hi - 1
return true
```

The step “clean text by the rule” in the second line is part of the problem definition. In the word “Kayak” the first letter is a capital K; if letters are compared as they are, K ≠ k is assumed and the right answer is missed. That is why a rule is fixed before the algorithm runs. The rule in the example is: convert all letters to lowercase and discard spaces and punctuation. “Kayak” thus becomes “kayak.”



Outside in: k = k, a = a; no comparison needed for the middle y → ⌊5 / 2⌋ = 2 comparisons

Figure 4.1.11: Comparison from both ends toward the center in the word “kayak”: k with k, a with a. The middle y is not compared with itself; for 5 letters, ⌊5 / 2⌋ = 2 comparisons are enough.

Figure 4.1.11 shows the compared pairs with arcs. lo and hi are the **indices** of the two letters being compared, that is, their positions in the text. Indices start at 0: in the word “kayak,” text[0] = 'k' and text[4] = 'k'. The trace of the algorithm is in Table 4.1.5.

Comparing	lo	hi	lo < hi	text[lo]	text[hi]	Decision
1	0	4	0 < 4 → true	'k'	'k'	same, move inward

Comparing	lo	hi	lo < hi	text[lo]	text[hi]	Decision
2	1	3	1 < 3 → true	'a'	'a'	same, move inward
—	2	2	2 < 2 → false	—	—	loop ends → return true

Table 4.1.5: Trace table of the palindrome algorithm for “Kayak.” The cleaned text is “kayak,” length 5; initially lo = 0, hi = 4.

At the third check, lo and hi are both 2; we have reached the middle letter. Since $2 < 2$ is false, the loop ends and true is returned. For a text of length n the loop runs at most $\lfloor n / 2 \rfloor$ times. The ⌊ ⌋ symbol means the number is rounded down: $\lfloor 5 / 2 \rfloor = \lfloor 2.5 \rfloor = 2$. In a text of even length, for example the word “abba,” lo and hi cross in the middle: after 2 comparisons lo = 2, hi = 1 and the loop ends again.

The algorithm stops even earlier on texts that are not palindromes. For “Kayık,” the first comparison finds k and k the same. In the second comparison a and ı differ, and false is returned; the middle y is never looked at.

Example 4.1.4

Is the sentence “Ey Edip Adana'da pide ye” (Turkish for “Hey Edip, eat pide in Adana”) a palindrome? How many comparisons are made?

First the cleaning rule is applied: letters are lowercased, and spaces and the apostrophe are discarded. The result is the 19-letter text “eyedipadanadapideye.” Initially lo = 0, hi = 18.

Comparing	lo	hi	lo < hi	text[lo]	text[hi]	Decision
1	0	18	0 < 18 → true	'e'	'e'	same, move inward
2	1	17	1 < 17 → true	'y'	'y'	same, move inward
3	2	16	2 < 16 → true	'e'	'e'	same, move inward
4	3	15	3 < 15 → true	'd'	'd'	same, move inward
5	4	14	4 < 14 → true	'i'	'i'	same, move inward
6	5	13	5 < 13 → true	'p'	'p'	same, move inward
7	6	12	6 < 12 → true	'a'	'a'	same, move inward
8	7	11	7 < 11 → true	'd'	'd'	same, move inward
9	8	10	8 < 10 → true	'a'	'a'	same, move inward
—	9	9	9 < 9 → false	—	—	loop ends → return true

In all nine comparisons the letters match; lo and hi meet at the middle letter, the 'n' at index 9. The sentence is a palindrome, and $\lfloor 19 / 2 \rfloor = 9$ comparisons were made. Without cleaning, the first comparison would treat capital 'E' and lowercase 'e' as different, and the answer would come out wrong.

Common mistake

Forgetting the letters I and İ when lowercasing Turkish text. In many programming languages, lowercasing applies the English rule unless a locale is specified, and turns “I” into “i.” In Turkish, however, the lowercase of “I” is “ı” and the lowercase of “İ” is “i.” The word “Irmak” becomes “irmak” under the English rule, and palindrome, search, or sorting results come out wrong. When the cleaning rule is written, the language of the text must be taken into account.

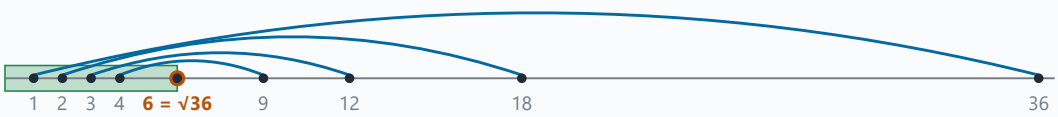
4.1.3.5 Prime numbers

A number greater than 1 that has no positive **divisor** other than 1 and itself is called a **prime number**. 2, 3, 5, 7, 11, and 13 are prime; 9 = 3 × 3 and 21 = 3 × 7 are not. The problem is to report whether a given number n is prime.

The definition directly yields an algorithm: try dividing n by every number from 2 to n − 1. If a number that divides it with no remainder is found, n is not prime. The remainder is computed with the `%` operation. *29 % 5 = 4*, because *29 = 5 × 5 + 4*. If *n % i = 0*, then i divides n exactly. This method is correct but does more work than necessary. The algorithm given here answers the same question with far fewer trials.

```
read n
if n < 2 then print "not prime" and stop
i = 2
while i × i ≤ n
    if n % i == 0 then print "not prime" and stop
    i = i + 1
print "prime"
```

The algorithm has two important properties. First, the search stops as soon as a divisor is found; there is no need to look at the remaining candidates. Second, the loop runs as long as the condition *i × i ≤ n* holds, that is, as long as i does not exceed √n. The reason for this **square-root bound** is the following: if n splits into two factors a × b, then a and b cannot both be greater than √n. If both were larger, their product would exceed n. So if n has a divisor greater than 1, it also has one that is less than or equal to √n.



36 = 1×36 = 2×18 = 3×12 = 4×9 = 6×6: the smaller of each pair is ≤ √36 → trying i = 2…6 is enough

Figure 4.1.12: The divisor pairs of 36: 1 × 36, 2 × 18, 3 × 12, 4 × 9, 6 × 6. The smaller member of each pair is less than or equal to √36 = 6; so trying the range i = 2, …, 6 is enough.

Figure 4.1.12 shows this observation for 36. Each arc connects a divisor pair. The left ends of the arcs are always in the green region, that is, to the left of 6 or at it. Writing the condition as $i \times i \le n$ instead of using the square root is a deliberate choice: integer multiplication replaces a fractional square-root computation, and there is no rounding error.

i	i × i ≤ n	n % i	Decision
2	4 ≤ 29 → true	29 % 2 = 1	not a divisor
3	9 ≤ 29 → true	29 % 3 = 2	not a divisor
4	16 ≤ 29 → true	29 % 4 = 1	not a divisor
5	25 ≤ 29 → true	29 % 5 = 4	not a divisor
6	36 > 29 → false	—	loop ends → prime

Table 4.1.6: Trace table of the prime algorithm for n = 29. Each row is one loop check.

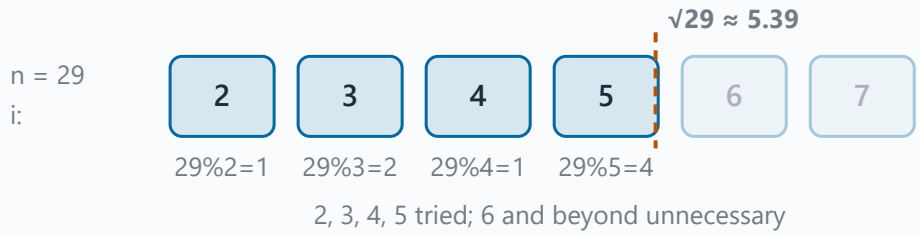


Figure 4.1.13: The divisors tried for n = 29. The dashed line is the bound √29 ≈ 5.39: only 2, 3, 4, and 5 are tried; 6 and beyond are unnecessary.

Table 4.1.6 and Figure 4.1.13 show the same run in two forms. 29 is divisible by none of the four candidates. When i = 6, *36 ≤ 29* is false and the loop ends. Since no divisor was found, “prime” is printed. The method taken from the definition would try 27 candidates, from 2 to 28; the square-root bound reduces this to 4. Had n been 21, *21 % 3 = 0* would be found at i = 3, “not prime” would be printed, and the algorithm would end after two trials.

Example 4.1.5

Is 221 prime? How many `%` operations does the algorithm perform?

221 is odd, it is not divisible by 3 because its digits sum to 5, and it does not end in 5. At first glance it may look prime. Since √221 ≈ 14.87, the algorithm tries at most up to i = 14.

i	i × i ≤ n	n % i	Decision
2	4 ≤ 221 → true	221 % 2 = 1	not a divisor
3	9 ≤ 221 → true	221 % 3 = 2	not a divisor
4	16 ≤ 221 → true	221 % 4 = 1	not a divisor
5	25 ≤ 221 → true	221 % 5 = 1	not a divisor

i	i × i ≤ n	n % i	Decision
6	36 ≤ 221 → true	221 % 6 = 5	not a divisor
7	49 ≤ 221 → true	221 % 7 = 4	not a divisor
8	64 ≤ 221 → true	221 % 8 = 5	not a divisor
9	81 ≤ 221 → true	221 % 9 = 5	not a divisor
10	100 ≤ 221 → true	221 % 10 = 1	not a divisor
11	121 ≤ 221 → true	221 % 11 = 1	not a divisor
12	144 ≤ 221 → true	221 % 12 = 5	not a divisor
13	169 ≤ 221 → true	221 % 13 = 0	divisor found → not prime

At i = 13 the remainder is 0: *221 = 13 × 17*. The algorithm prints “not prime” and stops. 12 candidates, from 2 to 13, have been tried, and the `%` operation has been performed 12 times. The smaller factor, 13, is below the √221 bound and so is caught; the larger one, 17, is never reached.

Common mistake

Writing the square-root bound as `i × i < n`, leaving out equality. This error shows up on perfect squares. For n = 49, when i = 7, *49 < 49* is false; the loop ends without trying 7. Since no number from 2 to 6 divides 49, the algorithm wrongly says 49 is “prime.” Perfect squares such as 4, 9, 25, and 49 are good test inputs for this bound.

4.1.4 Verifying an algorithm

That an algorithm has been written does not show that it works. Correctness is tested in two ways. The first is to run the algorithm on small inputs line by line, as a computer would. The second is to try the algorithm's behavior on different kinds of inputs. The two methods complement each other.

4.1.4.1 Hand tracing and the trace table

Running an algorithm on paper, applying each line in order, is called **hand tracing**. During a trace, the values of the variables are not kept in the head but written in a table. Rows show time, columns show variables. Such a table is called a **trace table**. All the tables in the previous section are trace tables.

Definition 4.1.5

A **trace table** is a table in which the value of each variable at each step is recorded while an algorithm is run by hand. Each column corresponds to a variable or a condition being checked, and each row to a step or a loop iteration.

The most important decision when preparing a trace table is what a row represents. Is the variable's value at the start of the iteration written, or its value at the end? If the two are mixed, the table becomes misleading. That is why this decision is stated explicitly in the table's caption or below it.

Example 4.1.6

How is the factorial algorithm traced for n = 4? In the table, the i column shows the value at the start of the iteration and the result column the value at the end.

State	i	result	i ≤ n
start	2	1	—
iteration 1	2	2	true
iteration 2	3	6	true
iteration 3	4	24	true
exit	5	24	5 ≤ 4 false

The loop runs 3 times and 24 is printed; since *4! = 24*, the result is correct. The loop invariant can also be checked on every row of the table: at the start of iteration 1, with i = 2, result is 1 = 1!; at the start of iteration 2, with i = 3, result is 2 = 2!.

The trace table is used not only to show correctness but also to find errors. When an algorithm gives a wrong result, the expected and actual values are compared row by row. The first row where the two values diverge shows where the error is. For example, if the condition of the factorial algorithm was mistakenly written `i < n`, then for n = 4 the table proceeds correctly up to iteration 2. When i = 4 the condition comes out false and iteration 3 is never performed. The location of the error is the loop condition.

Common mistake

Applying, during the trace, what the algorithm is supposed to do rather than what it says. Because the person tracing knows the goal of the algorithm, they unknowingly “correct” an overlooked line and fail to see the error. Hand tracing works when the lines are applied mechanically, exactly as written.

4.1.4.2 Test cases

A trace table shows that an algorithm works for one input. A single successful trial, however, does not prove that the algorithm is correct for every input. The off-by-one error in the factorial algorithm is invisible when tried with n = 1; the algorithm gives the correct answer, 1. That is why an algorithm is tested with a few carefully chosen inputs. A pair consisting of an input and the output expected for it is called a **test case**.

Definition 4.1.6

A **test case** is a pair consisting of an input to be given to an algorithm and the output expected for that input. Test cases fall into four kinds: **normal** (common use), **boundary** (values at the edge of the constraint), **invalid** (values outside the constraint), and **large** (values that expose scale problems such as overflow or slowness).

The expected output of a test case is written before the algorithm is run. Running the algorithm first and writing its result into the “expected” column makes the test meaningless. The expected value must come from the problem definition and from a hand computation.

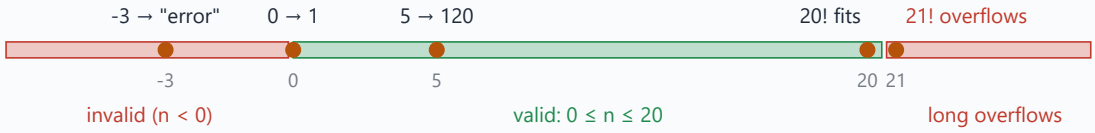


Figure 4.1.14: The number line for the factorial input. Negative values are invalid (−3 → “error”); 0 is the lower bound (0! = 1); 5 is a normal input (120); 0 to 20 is valid; from 21 on, the result does not fit in the long type.

Figure 4.1.14 shows the regions of the factorial input. At least one test is chosen from each region; the boundaries between regions are tried separately, because errors tend to hide at boundaries. Table 4.1.7 gives such a set of tests.

Kind	Goal	Input	Expected
Normal	Verify common use	n = 5	120 (the loop runs 4 times)
Boundary	Try the lower bound	n = 0	1 (the loop never runs)
Boundary	A second case where the loop never runs	n = 1	1 (the loop never runs)
Boundary	Try the upper bound	n = 20	2,432,902,008,176,640,000 (the loop runs 19 times)
Invalid	See the error behavior	n = −3	“error”

Kind	Goal	Input	Expected
Large	See the overflow	n = 21	51,090,942,171,709,440,000 → does not fit in long; the constraint check must give “error”

Table 4.1.7: Test cases for the factorial algorithm (constraint 0 ≤ n ≤ 20). The expected values were computed with Python's unbounded integers.

The last row of the table shows a special case. The value of 21! is itself correct, but Java's 64-bit `long` type can hold at most 9,223,372,036,854,775,807. 21! exceeds this limit by about a factor of 5.5. On a computation that exceeds the limit, Java does not report an error; the result silently wraps around. When the algorithm is run in Java with `long`, 21! gives a negative, meaningless number such as −4,249,290,049,419,214,848. A value exceeding the limit of the data type that holds it is called **overflow**.

This test reveals a gap in the algorithm: the upper bound of the constraint is never checked. The constraint “0 ≤ n ≤ 20” written in the problem definition must also be reflected in the algorithm. The corrected algorithm checks both bounds together before it starts processing the input:

```
read n
if n < 0 or n > 20 then print "error" and stop
result = 1
i = 2
while i ≤ n
    result = result × i
    i = i + 1
print result
```

In this version, when n = 21 is entered the loop is never entered and “error” is printed. As we see, testing does not merely find the bug; it improves the algorithm itself.

Common mistake

Testing only with normal inputs. Three tests for n = 5, n = 6, and n = 7 try the same region three times and show nothing new. n = 0, n = 20, n = −3, and n = 21, however, each answer a different question. A good test set is measured not by the number of tests but by the regions it covers.

4.1.5 Algorithm analysis

For a problem there is often more than one correct algorithm. To choose among them, we need to measure how much work the algorithms do. This measurement is called **algorithm analysis**. The goal of analysis is not to measure time in seconds; that time varies with the computer, the

programming language, and the other programs running at the moment. The goal is to find how the work done changes with the size of the input.

4.1.5.1 One problem, two algorithms

The problem is to find the sum of the integers from 1 to n: *1 + 2 + … + n*. The first algorithm that comes to mind (A) adds the numbers one by one with a loop:

```
sum = 0
i = 1
while i <= n
    sum = sum + i
    i = i + 1
```

The second algorithm (B) uses a formula attributed to Gauss and finds the result in a single step:

```
sum = n * (n + 1) / 2
```

Where the formula comes from is seen with a short computation. The sum is written twice, once from start to end and once from end to start:

> T = 1 + 2 + 3 + … + n
>
> T = n + (n − 1) + (n − 2) + … + 1

If the numbers lined up one under the other are added, every pair totals *n + 1*: *1 + n*, *2 + (n − 1)*, … There are n pairs in all. So *2T = n × (n + 1)*, that is, *T = n × (n + 1) / 2*. For n = 10 this gives *10 × 11 / 2 = 55*; the loop computation of *1 + 2 + … + 10* is also 55.

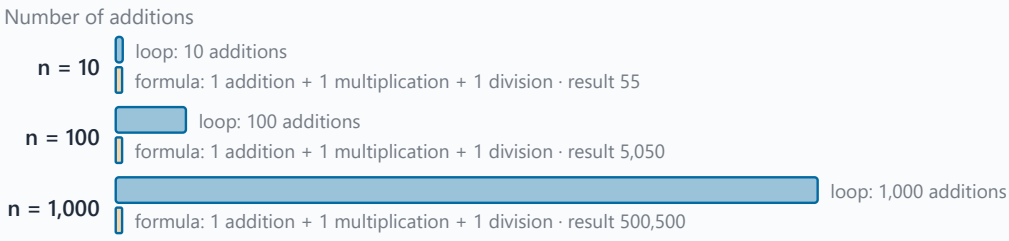


Figure 4.1.15: The number of operations the two algorithms perform. As n grows, the loop's number of additions grows with n (10, 100, 1,000); the formula's operation count stays constant, independent of n.

n	Result	A: additions in the loop	B: operations in the formula
10	55	10	1 multiplication, 1 addition, 1 division

n	Result	A: additions in the loop	B: operations in the formula
100	5,050	100	1 multiplication, 1 addition, 1 division
1,000	500,500	1,000	1 multiplication, 1 addition, 1 division
1,000,000	500,000,500,000	1,000,000	1 multiplication, 1 addition, 1 division

Table 4.1.8: Result and operation count of the two algorithms for 1 + 2 + … + n. The results were computed by both methods and came out equal.

Figure 4.1.15 and Table 4.1.8 show that both algorithms are correct, but that the work they do grows very differently. In A the loop runs n times; each iteration performs one addition and one counter increment. When n grows a thousandfold, A's work grows a thousandfold. B, whatever n is, performs one addition ($n + 1$), one multiplication, and one division. For the sum of a million numbers, A runs a million iterations; B still finishes with three operations.

4.1.5.2 Correctness and efficiency

The summation example shows that two separate criteria are used when evaluating algorithms. **Correctness** means that the algorithm produces the expected result. **Efficiency** is how many resources the algorithm uses to reach that result. The two most important resources are time and memory.

Definition 4.1.7

Time complexity describes how the number of basic operations an algorithm performs changes with the **input size**. **Space complexity** describes how the amount of extra memory the algorithm uses, beyond the input itself, changes with the input size.

The input size is usually denoted by the letter n and chosen according to the nature of the problem. In an algorithm that searches or sorts an array, n is the number of elements in the array. In an algorithm that computes a factorial or a sum, n is the number entered itself. The basic operation is also chosen according to the problem. In searching and sorting algorithms, we usually count the number of **comparisons** between two elements, because comparisons make up most of the work.

Different inputs of the same size may require different amounts of work. If the value searched for in an array is the first element, the search ends immediately; if it is the last element, the whole array is examined. That is why three cases are considered separately. The **best case** is the input on which the algorithm does the least work. The **worst case** is the input on which it does the most work. The average case is the amount of work expected on random inputs. In practice the worst case is used most: it guarantees that the algorithm will be no slower than this bound on any input.

4.1.5.3 Big-O notation

Counting operations exactly is often unnecessary. An algorithm performing 3n + 2 operations and one performing 5n grow in the same way: when n doubles, the work of both roughly doubles. What matters is not the coefficients but the shape of the growth. **Big-O notation** is a way of writing this shape concisely.

Definition 4.1.8

Big-O notation writes the fastest-growing term of an algorithm's operation count as n grows, dropping its coefficient. An algorithm performing 3n + 2 operations is O(n), and one making $n(n - 1)/2$ comparisons has complexity $O(n^2)$. An algorithm that performs a constant number of operations, independent of n, has complexity O(1).

The complexities met most often in this chapter are the following. **O(1)** is constant time: the Gauss formula performs three operations whatever n is. **O(log n)** is logarithmic time: at each step the problem is cut in half; binary search works this way. **O(n)** is linear time: each element is looked at once; summing with a loop and linear search are like this. **O(n log n)** is the complexity of merge sort. **$O(n^2)$** is quadratic time: each element is compared with most of the other elements; selection, bubble, and insertion sort are like this in the worst case.

The expression $\log_2 n$ is the answer to the question "to what power must 2 be raised to get n?" Since $2^{10} = 1{,}024$, $\log_2 1{,}024 = 10$. There is also a more intuitive reading: $\log_2 n$ says how many times n must be halved to get down to 1. The number 1,000,000 falls below 1 after being halved 20 times. That is why logarithmic algorithms do little work even on very large inputs.

n	Binary search ($\lfloor \log_2 n \rfloor + 1$ iterations)	Linear search (n)	Selection sort (n(n – 1)/2)	n^2
10	4	10	45	100
100	7	100	4,950	10,000
1,000	10	1,000	499,500	1,000,000
1,000,000	20	1,000,000	499,999,500,000	1,000,000,000,000

Table 4.1.9: Operation counts of different complexities as the input size n grows. The binary search column is the number of iterations for a value that is not found in a sorted array of n elements, and was counted by running the algorithm.

Table 4.1.9 shows the difference in numbers. At n = 10 the difference among the three columns is small; which method is chosen hardly matters. At n = 1,000,000, binary search finishes in 20 iterations, while linear search makes a million comparisons. Selection sort makes about five hundred billion comparisons. A computer that performs a billion comparisons per second finishes binary search in less than a millionth of a second, and selection sort in about eight minutes.

Efficiency is not always the only criterion. The method that looks fastest on small data is not always the easiest to understand. To sort a ten-element list, it is often wiser to choose a simple method that is easy to understand and verify than to write a complicated algorithm. As the input grows, the balance tips in favor of efficiency.

Common mistake

Mistaking Big-O notation for exact running time. It cannot be said that an O(n) algorithm is always faster than an $O(n^2)$ algorithm. The notation drops coefficients: an algorithm that performs 1,000n operations is O(n), but for n = 100 it performs 100,000 operations. An algorithm that performs n^2 operations finishes the same n with 10,000. Big-O tells which algorithm will pull ahead once n is large enough.

4.1.6 Searching

The task of finding out whether a particular value is present in a collection of data, and where it is if so, is called **searching**. Finding a phone number in a directory, a number in a student list, or a word in a dictionary are all searches. The value being sought is called the **target**. The algorithms here search for a target in an integer array. If the target is found they return its index; if not, they return −1. Since −1 is never a valid index, it is a safe marker meaning "absent."

The search algorithms are written in Java. In Java, `int[] arr` is an integer array; `arr[i]` is the element at index i and `arr.length` gives the number of elements. Indices start at 0; the last index of an array of n elements is $n - 1$. The `return` statement ends the method immediately and sends the value next to it back to the place that called the method.

4.1.6.1 Linear search

The simplest search method is to look at the elements in order from start to end. Each element is compared with the target; when a match is found, the search ends. Scanning the shelves one by one to find a book among books arranged in no order in a library is similar to this method.

Definition 4.1.9

Linear search is a search method that compares the elements of an array with the target in order, starting from the first element, and returns the index of the first matching element. If no element matches, −1 is returned.

```
int linearSearch(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target) return i;
    }
    return -1;
}
```

The method takes two parameters: the array to search and the target. The `for` loop increases i from 0 to *arr.length − 1*. On each iteration the comparison `arr[i] == target` is made. If they are equal, `return i` ends the method immediately; the rest of the loop does not run. If the loop ends without finding a match, control reaches the last line and −1 is returned.

Linear search has two important properties. First, the array does not need to be **sorted**; all elements are examined, in whatever order they are. Second, it stops at the first match. If the target occurs in the array more than once, the smallest index is returned.

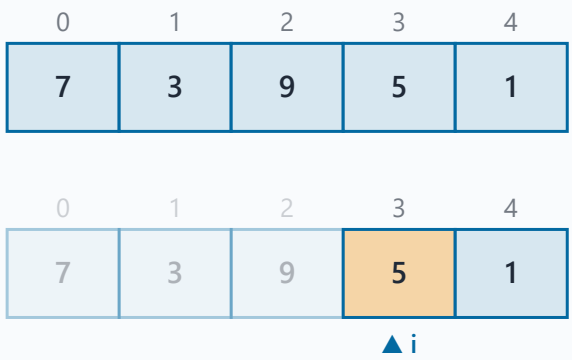


Figure 4.1.16: Linear search for 5 in the array {7, 3, 9, 5, 1}. On the left is the initial state; on the right, 5 has been found at i = 3. The faded cells are elements that were compared with the target and did not match.

i	i < arr.length	arr[i]	arr[i] == target	Result
0	0 < 5 → true	7	7 == 5 → false	next element
1	1 < 5 → true	3	3 == 5 → false	next element
2	2 < 5 → true	9	9 == 5 → false	next element
3	3 < 5 → true	5	5 == 5 → true	return 3

Table 4.1.10: Trace table of the call `linearSearch(arr, 5)`; arr = {7, 3, 9, 5, 1}.

As Table 4.1.10 shows, 5 is found at the fourth element and the method returns 3; 4 comparisons were made. Had the target been 4, all five elements would be tried, none would match, and −1 would be returned after 5 comparisons. The method compiled and run with JDK 21 also gives 3 for `linearSearch(arr, 5)` and −1 for `linearSearch(arr, 4)`.

The cost of linear search depends on the position of the target. In the best case the target is the first element and a single comparison is enough. In the worst case the target is the last element or is not in the array at all; n comparisons are made. That is why the time complexity of linear search is O(n). The only extra memory used is the variable i; the space complexity is O(1).

Common mistake

Reporting "not found" inside the loop. If `else return -1;` is added inside the loop, the method returns −1 right away when the first element is not the target and never looks at the remaining elements. −1 can be returned only after all elements have been tried, that is, after the loop ends.

4.1.6.2 Binary search

When looking up a name in a phone book, one does not start at the first page and turn pages one by one. One opens a page somewhere in the middle. If the name sought comes before the one on that page alphabetically, one moves to the front half; if it comes after, to the back half. With every look, half of the book is eliminated. This method depends on the book being in alphabetical order. Its computer counterpart is binary search.

Definition 4.1.10

Binary search is a search method that, in a sorted array, compares the target with the element in the middle of the **search range** and at each step eliminates the half of the range in which the target cannot be. If the range becomes empty, the target is not in the array.

```
int binarySearch(int[] arr, int target) {
    int lo = 0, hi = arr.length - 1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if (arr[mid] == target) return mid;
        if (arr[mid] < target) lo = mid + 1;
        else hi = mid - 1;
    }
    return -1;
}
```

The lines of the method work as follows:

1. `int lo = 0, hi = arr.length - 1;` : the search range is initially the whole array. lo is the first index of the range and hi the last.

2. `while (lo <= hi)` : the loop continues as long as at least one element remains in the range. If lo passes hi, the range is empty.
3. `int mid = lo + (hi - lo) / 2;` : the index in the middle of the range is computed. In Java, the quotient of two integers is an integer; the fractional part is discarded: *7 / 2 = 3*.
4. `if (arr[mid] == target) return mid;` : if the middle element is the target, the search ends.
5. `if (arr[mid] < target) lo = mid + 1;` : if the middle element is less than the target, then, because the array is sorted, all elements to the left of the middle are also less than the target. The left half is eliminated together with the middle.
6. `else hi = mid - 1;` : otherwise the middle element is greater than the target; the right half is eliminated together with the middle.
7. `return -1;` : if the range has become empty, the target is not in the array.

The elimination decision in lines 6 and 7 is the heart of the algorithm, and it rests entirely on the array being sorted. In a sorted array, if *arr[mid] < target*, then every one of *arr[0], ..., arr[mid]* is less than the target; none of them can be the target. That is why the middle itself is also eliminated: lo becomes *mid + 1*, not mid.

There is a reason the middle index is computed as `lo + (hi - lo) / 2` rather than `(lo + hi) / 2`. In mathematics the two expressions give the same result. On a computer, however, the `int` type holds at most 2,147,483,647. In a very large array, if lo = 1,500,000,000 and hi = 2,000,000,000, then `lo + hi` would exceed this limit and overflow. Tried with JDK 21, the expression `(lo + hi) / 2` gives a negative index such as −397,483,648. `lo + (hi - lo) / 2`, on the other hand, finds the correct result, 1,750,000,000, because the difference of the two indices is taken first and the difference never exceeds the limit.

In the example here the array has nine elements and the target is 34. Each iteration of the algorithm is shown in Table 4.1.11 and Figure 4.1.17.

Round	lo	hi	mid	arr[mid]	Decision
1	0	8	0 + (8 − 0) / 2 = 4	21	21 < 34 → lo = 5
2	5	8	5 + (8 − 5) / 2 = 6	30	30 < 34 → lo = 7
3	7	8	7 + (8 − 7) / 2 = 7	34	34 == 34 → return 7

Table 4.1.11: Trace table of the call `binarySearch(arr, 34)` ; arr = {3, 8, 12, 17, 21, 25, 30, 34, 40}.



Figure 4.1.17: Finding 34 by binary search, three iterations (in order). Iteration 1: range 0…8, mid = 4, arr[4] = 21. Iteration 2: range 5…8, mid = 6, arr[6] = 30. Iteration 3: range 7…8, mid = 7, arr[7] = 34; the target is found. Faded cells are eliminated elements.

In the first iteration the middle element, 21, is less than 34; 21 and the four elements to its left are eliminated. In the second iteration the middle of the remaining four elements is 30, which is again less than the target; 30 and the 25 to its left are eliminated. In the third iteration two elements remain. *7 + (8 − 7) / 2 = 7*, and arr[7] = 34 is found. Binary search has found the target in 3 iterations. Linear search would have made 8 comparisons to reach 34 in the same array.

4.1.6.3 Binary search: when the target is absent

The aspect of binary search that is easiest to misunderstand is how it stops when the target is not in the array. To see this, we search for 19 in the eight-element array {3, 8, 12, 17, 21, 25, 30, 40}. 19 is not in the array; if it were, it would sit between 17 and 21, at position 4.

Round	lo	hi	mid	arr[mid]	Decision
1	0	7	0 + (7 − 0) / 2 = 3	17	17 < 19 → lo = 4
2	4	7	4 + (7 − 4) / 2 = 5	25	25 > 19 → hi = 4
3	4	4	4 + (4 − 4) / 2 = 4	21	21 > 19 → hi = 3
—	4	3	—	—	4 ≤ 3 → false: the range is empty, return −1

Table 4.1.12: Trace table of the call `binarySearch(arr, 19)`; arr = {3, 8, 12, 17, 21, 25, 30, 40}.

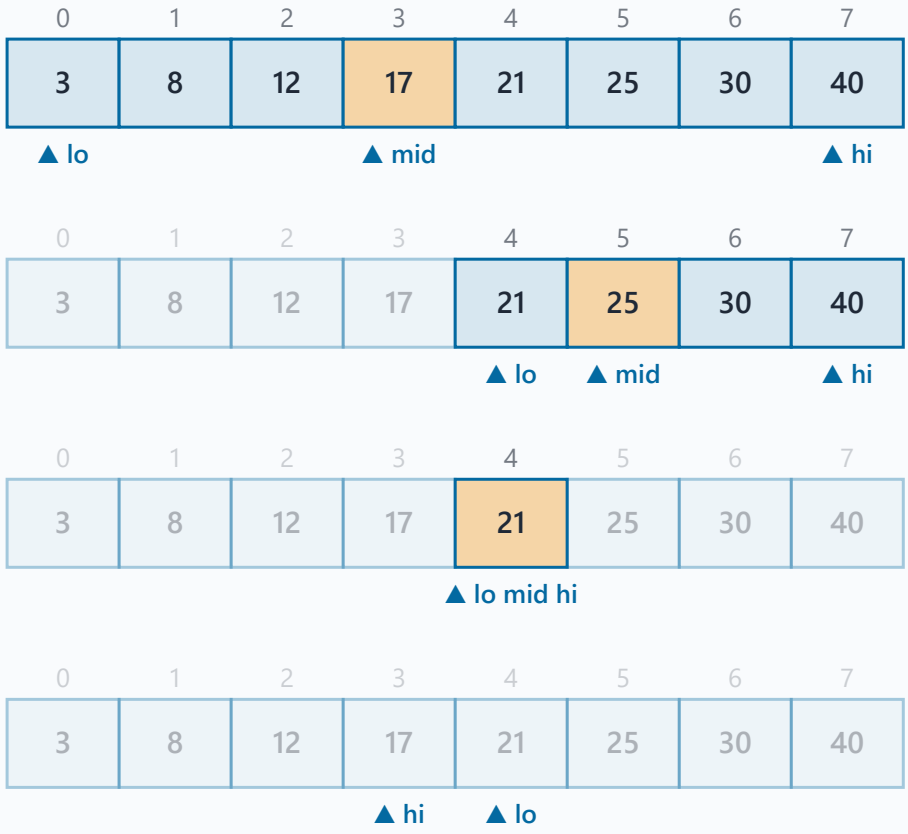


Figure 4.1.18: Binary search failing to find 19. Iteration 1: mid = 3, arr[3] = 17 < 19, the left half is eliminated. Iteration 2: mid = 5, arr[5] = 25 > 19, the right half is eliminated. Iteration 3: the range is a single element (4), arr[4] = 21 > 19, and hi becomes 3. In the final state hi is to the left of lo; the range is empty.

After the third iteration, lo = 4 and hi = 3. lo and hi have crossed; that is, lo has passed hi. Since the condition $4 \le 3$ is false, the loop ends and −1 is returned. The algorithm has given the answer "absent" in 3 iterations; it did not look at all eight elements one by one. The value of lo at the moment the loop ends is also meaningful: 4 is the position where the target would be found if it were in the array.

Java's own library uses this information. When the target is not found, the `java.util.Arrays.binarySearch` method returns $-(\textit{insertion point}) - 1$ instead of −1. For 19 in the same array, the value JDK 21 returns is −5: $-4 - 1 = -5$. Because the value is negative, the meaning "not found" is preserved; in addition, we learn at which position the target should be inserted.

Example 4.1.7

What happens if binary search is run for 5 on the unsorted array {7, 3, 9, 5, 1}?

Iteration 1: lo = 0, hi = 4, mid = 2, arr[2] = 9. Since 9 > 5, the right half is eliminated and hi becomes 1. The 5 is in fact on the eliminated side, at position 3. Iteration 2: lo = 0, hi = 1, mid = 0, arr[0] = 7. Since 7 > 5, hi becomes −1. $0 \le -1$ is false and −1 is returned. Although 5 is in the array, it has not been found. `binarySearch` run with JDK 21 also gives −1 for this array.

The algorithm raises no error and does not enter an infinite loop; it simply gives a wrong answer. This makes the bug more dangerous. Eliminating half is safe only if the array is sorted.

The cost of binary search is measured by the number of iterations. On each iteration the number of elements in the range falls to slightly less than half, because the middle is eliminated too. In an array of n elements the range becomes empty in at most $\lfloor \log_2 n \rfloor + 1$ iterations. In an eight-element array the bound is $\lfloor \log_2 8 \rfloor + 1 = 3 + 1 = 4$ iterations; the search for 19 ended in 3. In an array of a million elements the bound is 20 iterations. That is why the time complexity of binary search is O(log n). The extra memory requirement consists of the variables lo, hi, and mid; the space complexity is O(1).

Common mistake

Writing the loop condition as `lo < hi`, or updating the bounds as `lo = mid` and `hi = mid`. The first mistake means that when a single element remains in the range (lo = hi), that element is never looked at; at the end of the search for 3 in a 9-element array such a situation arises and the target is missed. The second mistake, because the middle is not eliminated, means the range sometimes never shrinks and the loop goes on forever. The correct form is the condition `lo <= hi` with the updates `mid + 1` and `mid - 1`.

4.1.6.4 Comparing search methods

The two search methods suit different conditions. Table 4.1.13 summarizes the differences.

Property	Linear search	Binary search
Sorted data	Not required	Required
Best case	1 comparison (target is the first element)	1 iteration (target is in the middle)
Worst-case time	O(n)	O(log n)
Extra memory	O(1)	O(1)
Structure	An array, or a data stream read from start to end	A sorted structure with random access
Suitable use	Little or unsorted data, a single search	Many searches on the same data

Table 4.1.13: Comparison of linear and binary search.

The “structure” row of the table points out a detail. Binary search depends on being able to reach the element in the middle of the array directly on each iteration. Reaching an element by its index in a single step, without passing through the ones before it, is called **random access**. Arrays have random access. In data that arrives in order from a file or a network, however, reaching the middle element requires reading all the earlier elements; for such data, linear search is the only option.

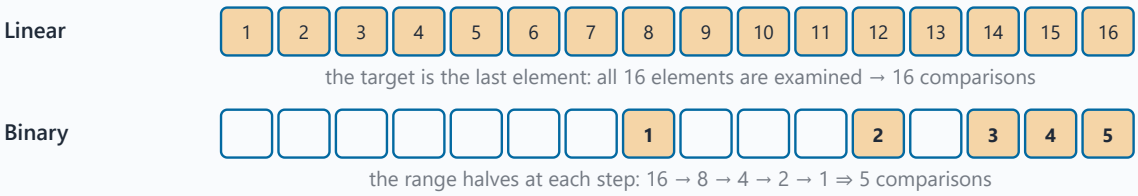


Figure 4.1.19: Searching for the last element in a sorted array of 16 elements. Linear search looks at all 16 elements. The elements binary search looks at are numbered in order: the range narrows 16 → 8 → 4 → 2 → 1 and 5 comparisons are enough.

Figure 4.1.19 shows the elements the two methods look at for the same target. When the array has 16 elements and the target is the last element, binary search looks at the 8th, 12th, 14th, 15th, and 16th elements in turn. Even in this small example the difference is more than threefold. The gap widens quickly as the array grows.

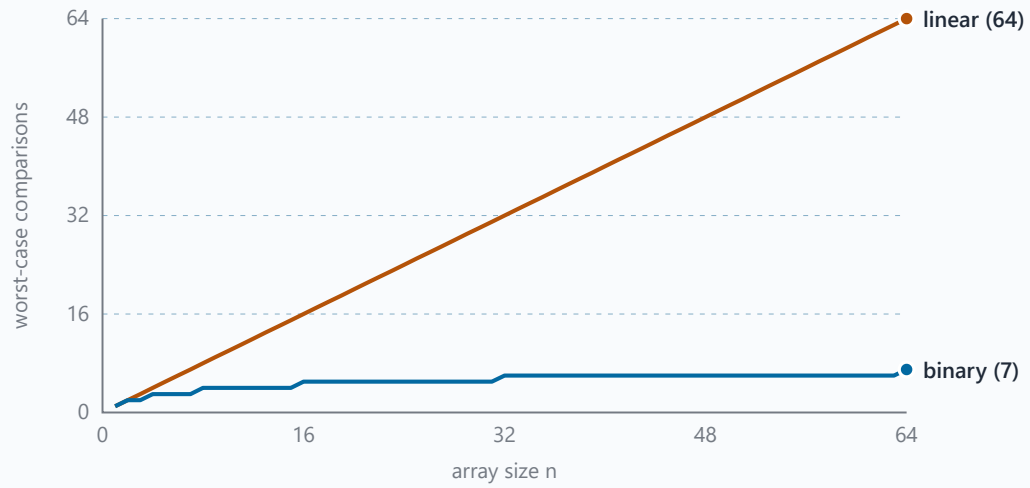


Figure 4.1.20: The number of comparisons in the worst case for a target that is not found. Linear search makes n comparisons (64 for n = 64); binary search makes $\lfloor \log_2 n \rfloor + 1$ iterations (7 for n = 64). The values were counted by actually running the two algorithms.

Figure 4.1.20 plots how the gap grows. The curve of linear search is a straight line; when n doubles, the number of comparisons doubles. The curve of binary search rises step by step, very slowly: when n doubles, the number of iterations increases by only one. For n = 1,000,000, linear search makes a million comparisons and binary search only 20.

This superiority has a price: binary search needs sorted data. If the data is not sorted, it must be sorted first. Sorting is a much more expensive task than searching itself; even the best general-purpose methods take O(n log n) time. For a single search, sorting first and then running binary search is more expensive than simply running linear search. If thousands of searches will be made on the same data, however, sorting once pays for itself quickly.

Example 4.1.8

500 searches will be made in an unsorted array of 1,000 elements. What are the worst-case comparison counts for linear search and for "selection sort first, then binary search"?

Linear search makes at most 1,000 comparisons per search: *500 × 1,000 = 500,000*. Selection sort makes *1,000 × 999 / 2 = 499,500* comparisons for 1,000 elements. After that, each binary search takes at most $\lfloor \log_2 1{,}000 \rfloor + 1 = 10$ iterations: *500 × 10 = 5,000*. The total is *499,500 + 5,000 = 504,500*. At this number of searches the two routes are almost equal. If the number of searches increases, the sorting route pulls ahead; if a faster sorting method is chosen, the gap opens much earlier.

4.1.7 Sorting

Arranging the elements of a collection in a particular order, for example from smallest to largest, is called **sorting**. Sorting is one of the things computers do most. Exam results are sorted by score, e-mails by date, products by price. Another value of sorting is that it makes later tasks easier: binary search works only on sorted data.

Four methods are compared here. The first three, selection, bubble, and insertion sort, are easy to understand and can be traced by hand. All three make $O(n^2)$ comparisons in the worst case. The fourth, merge sort, rests on a different idea and is much faster on large arrays. All methods sort the array from smallest to largest. Two elements standing side by side in an array are called **adjacent** elements.

4.1.7.1 Selection sort

A person sorting the cards in their hand can first find the smallest and put it at the front, then find the smallest among the rest and put it second. This natural method is called selection sort.

Definition 4.1.11

Selection sort is a sorting method that finds the smallest element in the unsorted part and swaps it with the element at the first position of that part. This step is repeated until one element remains in the unsorted part.

```
for i = 0 to n-2
    minIdx = i
    for j = i+1 to n-1
        if arr[j] < arr[minIdx] then minIdx = j
    swap arr[i] and arr[minIdx]
```

Each iteration of the outer loop is called a **pass**. At the start of pass i, the part 0…i − 1 of the array is sorted and holds the i smallest elements; the part i…n − 1 is unsorted. The variable `minIdx` is the index of the smallest element found so far. At the start, the first element of the unsorted part is taken as the smallest. The inner loop compares the rest of the part, one by one, with the element at minIdx; when a smaller one is found, minIdx is updated. When the inner loop finishes, the smallest element is swapped with the element at position i, and the sorted part grows by one element.

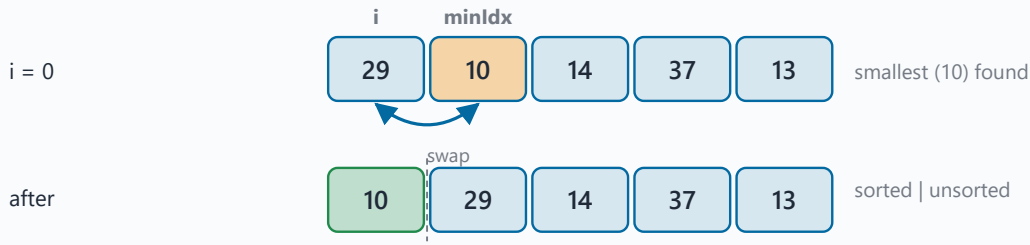


Figure 4.1.21: The first pass of selection sort: in the array {29, 10, 14, 37, 13} the smallest element is 10, and it is swapped with the 29 at position 0. At the end of the pass, 10 is in the sorted part; the remaining four elements are unsorted.

The example array here is {29, 10, 14, 37, 13}. The result of each pass is given in Table 4.1.14.

Pass	Unsorted part	Comparing	New values of minIdx	Swap	Array at end of pass
i = 0	0…4	4	10 (arr[1])	arr[0] ↔ arr[1]	[10, 29, 14, 37, 13]
i = 1	1…4	3	14 (arr[2]), 13 (arr[4])	arr[1] ↔ arr[4]	[10, 13, 14, 37, 29]
i = 2	2…4	2	unchanged	in place (no effect)	[10, 13, 14, 37, 29]
i = 3	3…4	1	29 (arr[4])	arr[3] ↔ arr[4]	[10, 13, 14, 29, 37]

Table 4.1.14: Trace table of selection sort for {29, 10, 14, 37, 13}. The “new values of minIdx” column shows, in order, the elements found smaller in the inner loop.

The rows of the table read as follows. At i = 0 the candidate smallest is 29. arr[1] = 10 is smaller and minIdx = 1; 14, 37, and 13 are not smaller than 10. 10 is moved to the front. At i = 1 the candidate is 29. First 14 is found (minIdx = 2), then 13 is smaller still (minIdx = 4). 13 and 29 are swapped. At i = 2 the candidate is 14, and the remaining 37 and 29 are not smaller; the smallest is already in place. In this pass the algorithm “swaps” arr[2] with itself; the operation has no effect. At i = 3, 29 and 37 are swapped and the array is sorted.

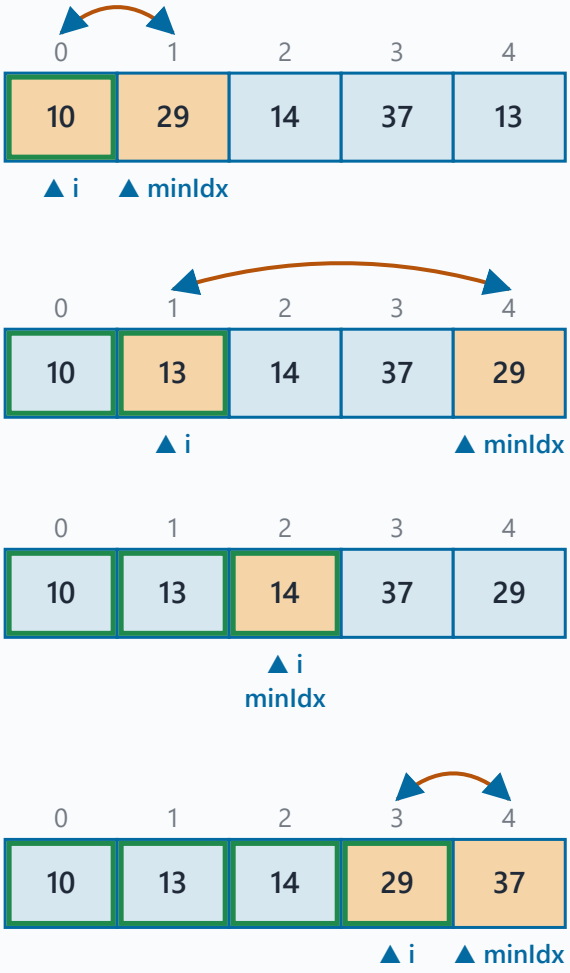


Figure 4.1.22: The end of the four passes of selection sort. i = 0: arr[0] and arr[1] are swapped. i = 1: arr[1] and arr[4] are swapped. i = 2: the smallest is already in place (minIdx = i). i = 3: arr[3] and arr[4] are swapped. Cells with a green frame are the sorted part.

The number of comparisons does not depend at all on the contents of the array. At i = 0 the inner loop makes *n − 1* comparisons, at i = 1 it makes *n − 2*, …, and in the last pass 1. The total:

> $(n - 1) + (n - 2) + \ldots + 1 = n(n - 1) / 2$

This is the Gauss formula with *n − 1* in place of n. For n = 5 it comes to *4 + 3 + 2 + 1 = 10* comparisons; the trace shown here also counted 10 comparisons. Even if the array is already sorted, the algorithm searches for the smallest again on every pass and still makes 10 comparisons. That is why selection sort is $O(n^2)$ in both the best case and the worst case. On the other hand, the number of swaps is small: at most one swap per pass, at most *n − 1* in total. Only a few helper variables are needed; the space complexity is $O(1)$.

Common mistake

Keeping the value of the smallest element in the inner loop instead of its index. The value is known, but if its place in the array is not known, the swap cannot be done. `minIdx` is an index; the comparison is made in the form `arr[j] < arr[minIdx]`. Another mistake is to put the swap inside the inner loop: in that case a swap is made every time a smaller element is found, and the number of swaps rises needlessly.

4.1.7.2 Bubble sort

Bubble sort compares only elements that stand side by side. In one pass the array is scanned from left to right, and every adjacent pair that is in the wrong order is swapped. Large values rise toward the right on every pass, like bubbles in water.

Definition 4.1.12

Bubble sort is a sorting method that compares adjacent elements from left to right and swaps every pair that is in the wrong order. Passes are repeated until a pass makes no swap at all.

```
round = 0
repeat
    swapped = false
    for j = 0 to n-2-round
        if arr[j] > arr[j+1] then
            swap arr[j] and arr[j+1]
            swapped = true
    round = round + 1
until swapped is false
```

At the end of a pass, the largest of the remaining elements has definitely reached the far right. The reason is this: wherever the largest element is, it wins every comparison with its right-hand neighbor and moves one place to the right. That is why each pass is one element shorter than the previous one. The variable `round` holds the number of completed passes and shortens the bound of the inner loop to $n - 2 - round$.

The variable `swapped` provides the algorithm's **early stopping** property. It is set to false at the start of every pass; if a swap happens in the pass, it becomes true. If no swap has happened in a pass, all adjacent pairs are in the right order, that is, the array is sorted. In that case no further pass is made and the algorithm stops. Writing the loop in the form “repeat … until” shows that the condition is checked after the body: the body runs at least once.

Pass	j	Pairs compared (↔ swap)	Comparing	Array at end of pass	swapped
Pass 1	0…3	(5, 1)↔ (5, 4)↔ (5, 2)↔ (5, 8)	4	[1, 4, 2, 5, 8]	true
Pass 2	0…2	(1, 4) (4, 2)↔ (4, 5)	3	[1, 2, 4, 5, 8]	true
Pass 3	0…1	(1, 2) (2, 4)	2	[1, 2, 4, 5, 8]	false

Table 4.1.15: Trace table of bubble sort for {5, 1, 4, 2, 8}. The ↔ sign shows that the pair was swapped.

Table 4.1.15 shows the three passes on the array {5, 1, 4, 2, 8}. In pass 1, 5 moves to position 3 by swapping in turn with 1, 4, and 2. When it is compared with 8, the order is right; 8 is already the largest and is at the far right. At the end of the pass, 8 is definitely in its place. In pass 2, only 4 and 2 are swapped and the array becomes sorted. The algorithm does not yet know this: swapped is true, so one more pass is made. In pass 3 no swap happens; swapped stays false and the algorithm stops. In total *4 + 3 + 2 = 9* comparisons were made.

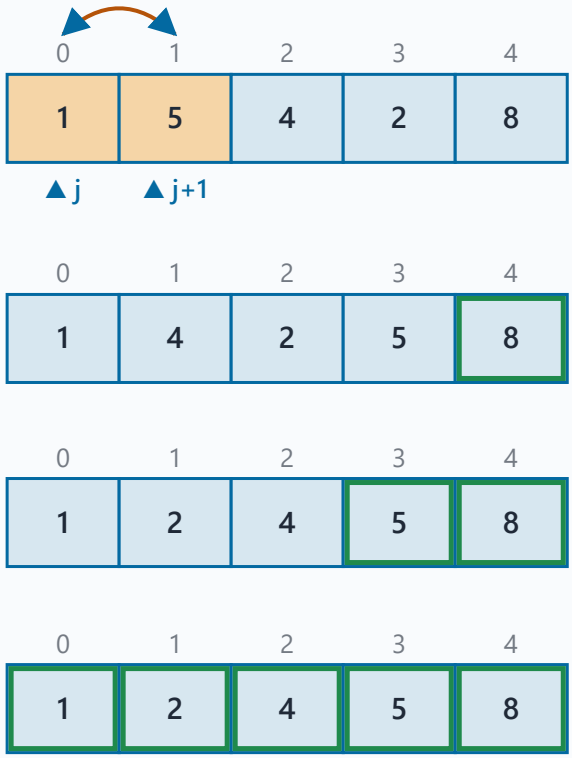


Figure 4.1.23: Four moments from bubble sort: the first swap in pass 1 (5 with 1); the end of pass 1 (8 in place); the end of pass 2 (5 also in place); the end of pass 3 (no swap happened, the array is sorted). Cells with a green frame are elements that have settled into their final places.

The cost of bubble sort depends on the input. In the worst case, when the array is in reverse order, every pair is swapped on every pass and $n - 1$ passes are needed. The number of comparisons, as in selection sort, is $n(n - 1)/2$; the time complexity is $O(n^2)$. In the best case the array is already sorted.

In the first pass $n - 1$ comparisons are made, no swap happens, and the algorithm stops: O(n). Without early stopping, the algorithm would make all the passes needlessly even on a sorted array. The extra memory is O(1).

Common mistake

Running the inner loop up to `n - 1 - round`. Because the comparison is made between `arr[j]` and `arr[j+1]`, the largest value of j is the left element of the last pair to be compared. In the first pass this index is $n - 2$. If j goes up to $n - 1$, `arr[j+1]` falls outside the array; in this case Java throws an `ArrayIndexOutOfBoundsException`.

4.1.7.3 Insertion sort

In card games, players usually arrange the cards in hand like this: when a new card is picked up, the right place is found among the cards already sorted in the hand, and the card is slipped in there. Insertion sort is exactly this method.

Definition 4.1.13

Insertion sort is a sorting method that always keeps the left part of the array sorted, takes the next element on the right (the **key**), and places it into the gap opened by shifting the elements greater than it one position to the right.

```
for i = 1 to n-1
    key = arr[i]
    j = i - 1
    while j >= 0 and arr[j] > key
        arr[j+1] = arr[j]
        j = j - 1
    arr[j+1] = key
```

The outer loop starts i at 1, because a part with one element (0…0) is already sorted. On each iteration the key is taken from arr[i]; the key's place can now be considered a “gap.” j starts at the element to the left of the key. The inner loop continues on two conditions: j must be inside the array ($j \geq 0$) and arr[j] must be greater than the key. As long as the condition holds, arr[j] is shifted one place to the right and j decreases by one; the gap moves to the left. When the loop stops, the gap is at position $j + 1$ and the key is written there.

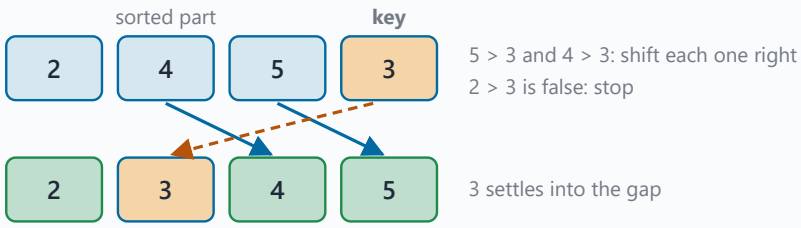


Figure 4.1.24: One step of insertion sort. The sorted part is {2, 4, 5} and the key is 3. Since 5 > 3 and 4 > 3, both shift one place right; since 2 > 3 is false, we stop. 3 settles into the gap: {2, 3, 4, 5}.

It matters that the check `j >= 0` comes first in the condition. When j = −1 this condition is false. The word “and” does not look at the second condition if the first is false; arr[−1] is never read. If the order of the two conditions is swapped, arr[−1] is read when the key is about to be placed at the very front, and the program fails.

Step	key	Comparisons	Key	Shifts	Index where placed	Array after
i = 1	3	4 > 3 ✓ (j = −1, stops)	1	1	0	[3, 4, 5, 1]
i = 2	5	4 > 5 ✗	1	0	2	[3, 4, 5, 1]
i = 3	1	5 > 1 ✓, 4 > 1 ✓, 3 > 1 ✓ (j = −1, stops)	3	3	0	[1, 3, 4, 5]

Table 4.1.16: Trace table of insertion sort for {4, 3, 5, 1}. ✓ is a comparison where the condition arr[j] > key came out true (the element is shifted), ✗ one where it came out false (the loop stops).

Table 4.1.16 is the trace of the array {4, 3, 5, 1}. At i = 1 the key is 3; since 4 > 3, 4 shifts right, j = −1, and the loop stops because it has reached the left end. 3 is placed at position 0. At i = 2 the key is 5; the 4 to its left is smaller, so its place is found with a single comparison and no shifting is done. At i = 3 the key, 1, is smaller than all the elements. 5, 4, and 3 shift right in turn and 1 is placed at the very front. In total *1 + 1 + 3 = 5* comparisons were made.

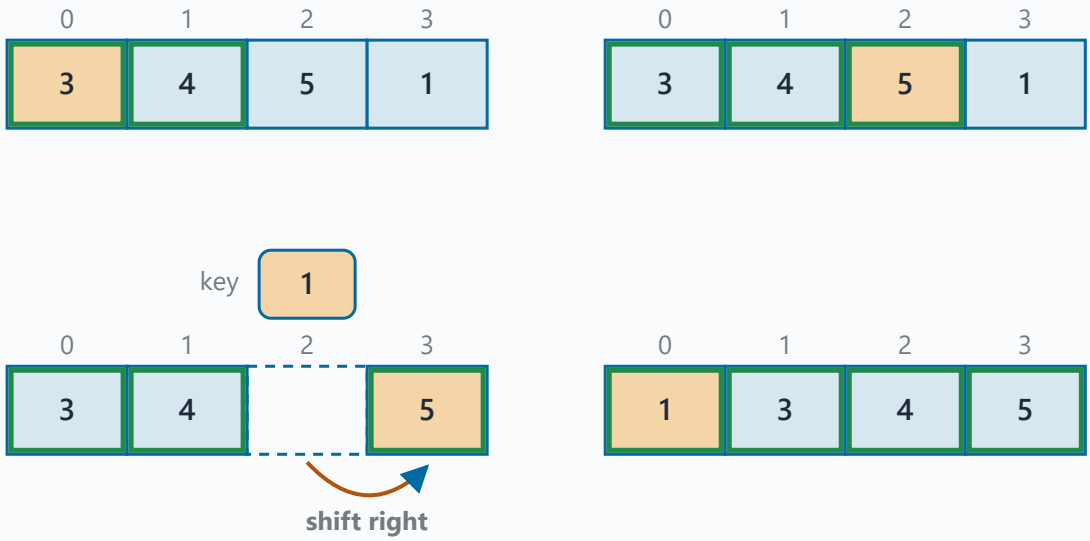


Figure 4.1.25: Four moments from insertion sort: at the end of i = 1, 3 has been placed; at the end of i = 2, 5 has been placed without shifting; at i = 3, for key 1, 5 shifts right; at the end of i = 3, 1 has been placed at the front and the array is sorted. Cells with a green frame are the sorted part.

The cost of insertion sort depends on how far the elements are "out of place." In the worst case the array is in reverse order; every key is shifted all the way to the front of the sorted part and $n(n-1)/2$ comparisons are made: $O(n^2)$. In the best case the array is already sorted; for each key a single comparison is made and there is no shifting. A total of $n-1$ comparisons: $O(n)$. The extra memory is just the key and j variables: $O(1)$.

This property makes insertion sort valuable in practice. It runs very fast on arrays with few elements or that are nearly sorted. In situations where new records are occasionally added to a sorted list, for example an alphabetical list to which a few students are added every day, insertion sort is a natural choice.

Common mistake

Starting to shift without saving the key in a variable. The first shift is the assignment `arr[i] = arr[i-1]`, and it erases the value in arr[i]. If the key has not been copied into the `key` variable beforehand, this value is lost. The situation is the same as the temporary variable in a swap: a value that is about to be overwritten is saved first.

4.1.7.4 Merge sort

The previous three methods place elements one at a time and make $O(n^2)$ comparisons. **Merge sort** rests on a different idea: merging two sorted arrays into a single sorted array is very cheap. The smallest element of two sorted arrays is the smaller of their two first elements. That element is taken,

and the same comparison is repeated for the remaining elements. Each comparison adds one element to the result; if the total number of elements is m, at most $m - 1$ comparisons are enough.

The algorithm uses this merge operation as follows. The array is split in the middle into two halves. Each half is sorted separately by the same method. Then the two sorted halves are merged. The halves are split again and again until they come down to arrays of one element; an array of one element is already sorted. Splitting a problem into smaller problems of the same kind and combining their solutions is called the **divide and conquer** approach.

```
merge(A, B):
    C = empty array
    i = 0, j = 0
    while i < A.length and j < B.length
        if A[i] ≤ B[j] then append A[i] to C, i = i + 1
        else append B[j] to C, j = j + 1
    append the remaining elements of A and B to C in order
    return C

sort(arr):
    if arr has at most 1 element return arr
    split arr in the middle into two halves: left and right
    return merge(sort(left), sort(right))
```

The `sort` operation calls itself again for smaller arrays. Such an operation, which is reused in its own definition, is called **recursive**. A detailed understanding of recursion is not needed for this chapter; what matters is that the splitting stops when it reaches arrays of one element, and that the merge at each level passes its result up to the next level.

Example 4.1.9

How is the array {38, 27, 43, 3, 9, 82, 10, 5} sorted by merge sort? How many comparisons are made?

The array is first split into the halves {38, 27, 43, 3} and {9, 82, 10, 5}, these into pairs, and finally into single elements. The merges are then done from the bottom up:

Two sorted parts merged	Result	Comparing
[38] + [27]	[27, 38]	1
[43] + [3]	[3, 43]	1
[27, 38] + [3, 43]	[3, 27, 38, 43]	3
[9] + [82]	[9, 82]	1
[10] + [5]	[5, 10]	1
[9, 82] + [5, 10]	[5, 9, 10, 82]	3
[3, 27, 38, 43] + [5, 9, 10, 82]	[3, 5, 9, 10, 27, 38, 43, 82]	7

Two sorted parts merged	Result	Comparing
Total		**17**

In the final merge, the parts {3, 27, 38, 43} and {5, 9, 10, 82} merge like this: 3 < 5 → 3; 27 > 5 → 5; 27 > 9 → 9; 27 > 10 → 10; 27 < 82 → 27; 38 < 82 → 38; 43 < 82 → 43. The left part is used up; the remaining 82 is appended without a comparison. 8 elements have been merged with 7 comparisons. Over all the merges, 17 comparisons are made in total. For the same array, selection sort would make *8 × 7 / 2 = 28* comparisons.

The cost of merge sort comes from the number of levels. Because the array is halved at every level, there are $log_2\ n$ levels; 3 levels for 8 elements. At each level all the elements go through a merge once, and at most n comparisons are made. The total number of comparisons is about $n \times log_2\ n$; the time complexity is O(n log n) in the best, average, and worst cases. The price is memory: the result of a merge is built in a separate array. So the extra memory is O(n).

4.1.7.5 Comparing sorting methods

Algorithm	Basic idea	Worst case	Best case	Extra space
Selection	Select the smallest and place it	$O(n^2)$	$O(n^2)$	O(1)
Bubble	Swap neighbors	$O(n^2)$	O(n) (with early stopping)	O(1)
Insertion	Insert the element into the sorted part	$O(n^2)$	O(n)	O(1)
Merge	Split, sort, merge	O(n log n)	O(n log n)	O(n)

Table 4.1.17: Comparison of the four sorting methods.

Table 4.1.17 summarizes the four methods in terms of complexity. Even methods with the same Big-O value actually make different numbers of comparisons. Table 4.1.18 counts them by running the three simple methods on the same five-element arrays.

Input (n = 5)	Selection	Bubble (with early stopping)	Insertion
Sorted [1, 2, 3, 4, 5]	10 / 0	4 / 0	4 / 0
Reverse sorted [5, 4, 3, 2, 1]	10 / 2	10 / 10	10 / 10
Shuffled [5, 1, 4, 2, 8]	10 / 2	9 / 4	7 / 4

Table 4.1.18: Numbers of comparisons and swaps on three five-element arrays (comparisons / swaps). In insertion sort the second number is the number of shifts; in selection sort only swaps that actually move elements are counted.

The table confirms the “best case” column of the complexity table with numbers. On a sorted array, selection sort still makes 10 comparisons; bubble and insertion sort finish the job with 4. On a reverse-sorted array all three methods make 10 comparisons, but selection sort gets by with just 2 swaps. That selection sort makes few swaps is an advantage when writing is expensive.

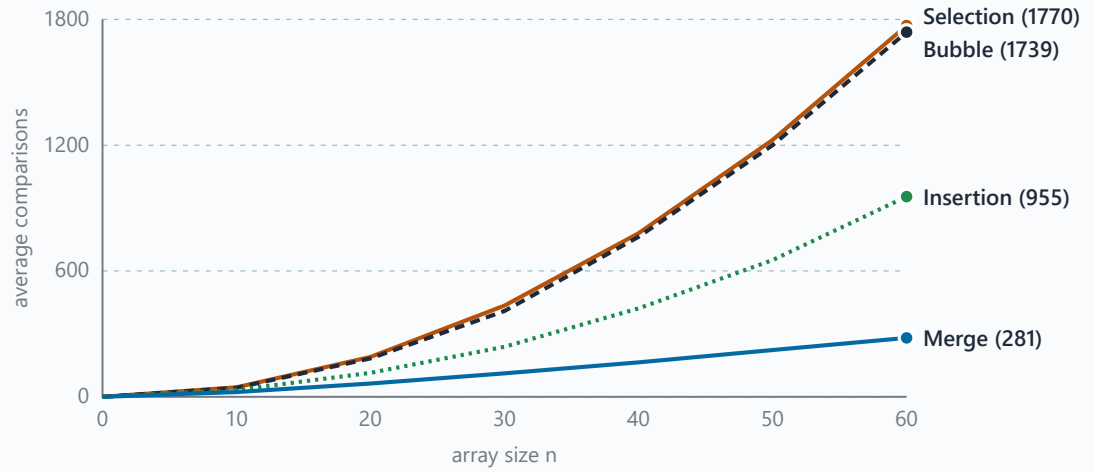


Figure 4.1.26: The average number of comparisons on random arrays. For each n, 20 random arrays were sorted with the four algorithms and the averages are plotted. At n = 60, selection made 1,770, bubble 1,739, insertion 955, and merge 281 comparisons.

Figure 4.1.26 shows the average behavior on random arrays. Selection and bubble sort make about $n^2 / 2$ comparisons; for n = 60, selection makes exactly $60 \times 59 / 2 = 1{,}770$. Insertion sort on average makes about half of that, around $n^2 / 4$. The reason is that in a random array each key passes over about half of the sorted part on average. Merge sort stays around $n \log_2 n$, and the gap opens quickly as n grows. In the computation for this chapter, an average over 2,000 random arrays gave, for n = 60, about 1,730 for bubble sort, 939 for insertion sort, and 281 for merge sort; the 20-array averages in the figure are close to these values.

In practice, ready-made library methods are used. Java's `java.util.Arrays.sort` method uses sophisticated methods that run in O(n log n) time. The method used on arrays of objects (Timsort) combines ideas from merge sort and insertion sort. On arrays of primitive types such as `int`, dual-pivot quicksort is used. Both methods switch to insertion sort on small pieces of the array, because insertion sort is fast on small and nearly sorted data. The value of learning the simple methods is that it lets us understand the reasons for these design decisions.

Seeing how sorting algorithms work complements the trace tables. The Sorting Algorithm Visualizer runs the four methods step by step on random, sorted, reverse-sorted and nearly sorted arrays and shows the comparison counts.

4.1.8 Summary

- **Problem solving** does not begin with code: the problem is understood, a small example is solved by hand, the method is written down, and only then is it coded. A **problem definition** consists of the goal, input, output, **constraint**, example, and error case.

- An **algorithm** is a finite sequence of well-defined steps for solving a problem; it takes zero or more inputs and produces at least one output. The steps of a good algorithm are definite, ordered, and feasible; the algorithm stops in a finite number of steps, gives the right result for every valid input, and uses resources efficiently.
- Algorithm design is a cycle: solve an example, find the general rule, write pseudocode, trace by hand, fix. **Pseudocode** writes an algorithm without tying it to a particular language; indentation shows the scope of conditions and loops.
- A **swap** of two variables takes three assignments and a **temporary variable**. The order of assignments in Fibonacci matters for the same reason: a value that is about to be overwritten is first used or saved.
- The starting value of an **accumulator**, which builds up a result step by step, is chosen according to the operation: 0 for a sum, 1 for a product. The **loop invariant** of the factorial algorithm is "at the start of each iteration, result = (i − 1)!", and it explains why the algorithm is correct.
- Palindrome checking moves from both ends toward the middle and makes at most ⌊n / 2⌋ comparisons. Primality checking searches for divisors only up to $\sqrt{n}$ ($i \times i \leq n$). Both algorithms stop early as soon as the result is known.
- **Hand tracing** and the **trace table** show that an algorithm works for one input. **Test cases** cover normal, boundary, invalid, and large inputs. A single successful test does not prove that an algorithm is correct for every input; the 21! test exposes overflow and a missing constraint check.
- **Time complexity** describes how the number of operations grows with the input size, and **space complexity** how the extra memory grows; both are written in **Big-O notation**: O(1), O(log n), O(n), O(n log n), $O(n^2)$. The sum 1 + 2 + … + n is found in O(n) time with a loop and in O(1) time with the Gauss formula.
- **Linear search** does not require sorting and makes n comparisons in the worst case: O(n). **Binary search** works only on sorted data, halves the range on every iteration, and makes at most $\lfloor \log_2 n \rfloor + 1$ iterations: O(log n). At n = 1,000,000 the difference is 1,000,000 against 20.
- **Selection sort** picks the smallest on every pass; it always makes n(n − 1)/2 comparisons but at most n − 1 swaps. **Bubble sort** swaps neighbors and stops on a pass with no swaps. **Insertion sort** inserts the key into the sorted part; it is very fast on nearly sorted data. All three are $O(n^2)$ in the worst case and O(1) in extra space.
- **Merge sort** splits the array in two, sorts the halves, and merges them; in every case it uses O(n log n) time and O(n) extra memory.

4.1.9 Key terms

The key terms of this chapter are listed below.

problem solving problem definition constraint algorithm definiteness finiteness correctness pseudocode indentation swap temporary variable factorial accumulator loop invariant

off-by-one error sliding window palindrome prime number divisor remainder hand tracing trace table test case invalid input overflow time complexity space complexity input size Big-O notation best case worst case search target linear search binary search search range random access sorting sorted adjacent pass selection sort bubble sort insertion sort key merge sort

4.1.10 Test Yourself

1. The algorithm that orders three numbers (first compare a–b, then b–c, finally a–b again) is run for a = 4, b = 9, c = 2. What are a, b, c after each comparison? How many swaps are made?

   Answer

   1st comparison: *4 > 9* is false; no change (4, 9, 2). 2nd comparison: *9 > 2* is true; b and c are swapped (4, 2, 9). 3rd comparison: *4 > 2* is true; a and b are swapped (2, 4, 9). The output is "2 4 9"; two swaps were made.

2. In the algorithm that sums the even numbers, `i = 1` was mistakenly written instead of `i = 2`; the increment is still `i = i + 2`. What is printed for n = 10? Which test catches the error?

   Answer

   The counter takes the values 1, 3, 5, 7, 9 and the odd numbers are summed: *1 + 3 + 5 + 7 + 9 = 25*. The correct answer is 30. The test n = 1 also shows the error: 0 is expected and 1 is found. The test n = 0 does not show the error, because in both versions the loop never runs and 0 is printed.

3. The prime algorithm is run for n = 97. Which values of i are tried, how many times is the % operation performed, and what is the result?

   Answer

   The 8 values from i = 2 to 9 are tried, because *9 × 9 = 81 ≤ 97* but *10 × 10 = 100 > 97*. The remainders are 1, 1, 1, 2, 1, 6, 1, 7 in order; none is 0. The % operation is performed 8 times and "prime" is printed.

4. The palindrome algorithm is run on the texts "Kalem" and "Kepek" (Turkish for "pencil" and

"dandruff"). How many comparisons are made for each, and what is the result?

Answer

Lowercased, "Kalem" becomes "kalem." In the first comparison text[0] = 'k' and text[4] = 'm' differ; after 1 comparison false is returned. For "Kepek," k with k, then e with e are the same; when lo = hi = 2 the loop ends. With 2 comparisons true is returned.

5. In the array {4, 8, 15, 16, 23, 42}, 16 is searched for first by linear search and then by binary search. How many comparisons or iterations does each method make? In binary search, what are lo, hi, and mid on each iteration?

Answer

Linear search looks at 4, 8, 15, and 16; after 4 comparisons it returns 3. Binary search: iteration 1: lo = 0, hi = 5, mid = 2, arr[2] = 15 < 16 → lo = 3. Iteration 2: lo = 3, hi = 5, mid = 4, arr[4] = 23 > 16 → hi = 3. Iteration 3: lo = hi = 3, mid = 3, arr[3] = 16 → 3 is returned. Binary search finds it in 3 iterations.

6. A value that is not present is searched for in a sorted array of 4,000 elements. How many iterations does binary search make at most? If the array grows to 8,000 elements, how does the worst-case work of linear search and of binary search change?

Answer

Since $2^{11} = 2{,}048 \le 4{,}000 < 4{,}096 = 2^{12}$, $\lfloor \log_2 4{,}000 \rfloor = 11$; binary search makes at most $11 + 1 = 12$ iterations. When the array doubles, linear search's work also doubles (from 4,000 to 8,000 comparisons). The number of iterations of binary search increases by only one, to 13.

7. The array [3, 1, 2, 4] is sorted with selection sort, bubble sort with early stopping, and insertion sort. How many comparisons does each method make? How many passes does bubble sort make?

Answer

Selection: $3 + 2 + 1 = 6$ comparisons; the contents of the array do not change the result. Bubble: in pass 1, (3, 1) and (3, 2) are swapped and (3, 4) is in the right order; 3 comparisons, and the array becomes [1, 2, 3, 4]. In pass 2, 2 comparisons are made, no swap happens, and the algorithm stops. In total 2 passes and 5 comparisons. Insertion: for key 1, 1 comparison (3 > 1, then j = −1); for key 2, 2 comparisons (3 > 2, 1 > 2 is false); for key 4, 1 comparison; a total of 4. Because the array is nearly sorted, insertion sort makes the fewest comparisons.

8. What are at least five test cases for the palindrome algorithm, and the expected result for each?

Which possible bug does each test catch?

Answer

A sample test set: the empty text → true (the loop never runs; the algorithm must not crash). A single letter, “a” → true. An even-length palindrome, “abba” → true (lo and hi cross in the middle). A palindrome with a capital letter, “Kayak” → true (comes out false if the cleaning rule is not applied). A text containing Turkish I, “Iı” → true (comes out false if I is converted to i). A text that is not a palindrome, “kayık” → false. A text whose last letter differs, “kayaz” → false (must stop at the first comparison).

9. An algorithm takes 2 seconds for n = 1,000. About how long is it expected to take for n = 2,000? How would the answer change if the algorithm were O(n), O(n^2), or O(log n)?

Answer

In an O(n) algorithm the work grows in proportion to n; when n doubles, the time roughly doubles: 4 seconds. In an O(n^2) algorithm the work grows with the square of n; it grows by a factor of $2^2 = 4$: about 8 seconds. In an O(log n) algorithm, doubling n increases $\log_2 n$ by only 1 (from about 10 to 11); the time increases by about 10% and stays around 2.2 seconds. These are rough estimates; Big-O ignores coefficients and smaller terms.

4.1.11 Further reading

- R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), 4.1 Performance and 4.2 Sorting and Searching: measuring operation counts by experiment, binary search, and the simple sorting methods with Java examples; the book's site (introcs.cs.princeton.edu) has chapter summaries and code.
- R. Sedgewick, K. Wayne, *Algorithms* (4th ed., Addison-Wesley, 2011), Chapter 2: compares selection, insertion, and merge sort with visual traces; the animations on the book's site (algs4.cs.princeton.edu) complement the trace tables in this chapter.
- A. Levitin, *Introduction to the Design and Analysis of Algorithms* (3rd ed., Pearson, 2012), Chapters 1–3: treats the notion of an algorithm, problem definition, and brute-force methods (selection and bubble sort, linear search) in plain language.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (4th ed., MIT Press, 2022), Chapter 2: gives the loop invariant and the correctness proof through insertion sort, and the analysis of merge sort, rigorously; for those who want to continue the subject at an advanced level.

4.1.12 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
2. A. Levitin, *Introduction to the Design and Analysis of Algorithms*, 3rd ed., Pearson, 2012.
3. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011. Book site.
4. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 4.1 Performance, 4.2 Sorting and Searching.
5. Oracle, *Java SE 21 API*: java.util.Arrays (sort, binarySearch).

PART II · CHAPTER 4.2

Flowcharts

When an algorithm is first put on paper, a drawing often comes before any text. Boxes show the steps and arrows show their order. Such a drawing is called a flowchart. This chapter introduces the symbols used in flowcharts and the rules of a good diagram. Then come the three kinds of flow that are the building blocks of every algorithm. In sequential flow, steps run one after another. In conditional flow, the path splits according to a question. In repetitive flow, a group of steps runs again and again. Each structure is executed step by step on small examples, and the value of each variable at each step is given in trace tables. Loop patterns such as the counter, the accumulator, and the sentinel value, as well as nested loops, early exit, and flags, are also treated separately. The last sections show how a flowchart is translated into Java code and which inputs a flowchart is tested with.

LEARNING OUTCOMES

Symbols
Recognize the basic flowchart symbols (start/end, process, decision, input/output, arrow, connector) and the additional symbols; choose the right symbol for each step.

Flow
Establish the order of operations with arrows; apply the rules of a readable diagram; describe the same algorithm with a flowchart and with pseudocode.

Control
Show sequential, conditional, and repetitive flow in a diagram; use the patterns of decision chain, compound condition, counter, accumulator, sentinel value, nested loop, post-test loop, early exit, and flag; execute a diagram step by step with a trace table.

Translation
Translate a diagram into pseudocode and into a Java program; split a large diagram into subprocesses; test a diagram with normal and boundary inputs and with inputs on which the loop runs zero, one, and many times.

4.2.1 The flowchart

An algorithm is a clear and finite sequence of steps for solving a problem. Steps can also be described in plain prose. But as the text grows longer, it becomes easy to lose track of which step comes after which, and under what conditions a step is skipped or repeated. A flowchart makes this order visible.

4.2.1.1 What is a flowchart?

Definition 4.2.1

A **flowchart** is a drawing that shows the steps of an algorithm with standard symbols and the order in which they run with arrows. The shape of a symbol tells the kind of step (start, process, decision, input/output), the short expression inside tells what the step does, and the arrow tells which step execution moves to.

An everyday example makes the definition concrete. An apartment building has a keypad lock on its door. The small computer inside the lock does the same job each time: it reads the code entered, checks whether it is correct, opens the door if it is, and sounds a warning otherwise. Figure 4.2.1 shows this job as a flowchart.

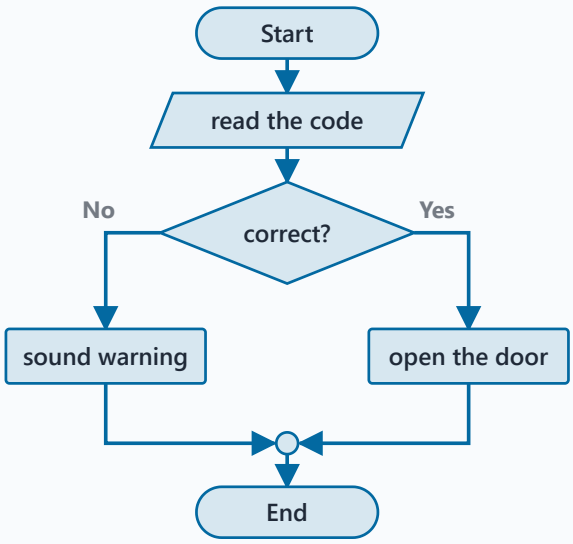


Figure 4.2.1: Flowchart of the door-code algorithm. Two paths leave the decision box: if the code is correct, "open the door" runs; otherwise "sound warning" runs. The two paths join at the small circle below and reach End.

The diagram is read from top to bottom. Execution begins at the round-ended box labeled *Start*. An arrow carries execution to the "read the code" box. Next comes the diamond-shaped decision box. It contains a question: "correct?" Depending on the answer, one of two arrows is followed. The path

along the Yes arrow opens the door; the path along the No arrow sounds a warning. Both paths finally meet at the same point and reach the *End* box.

This small drawing shows three basic properties of a flowchart. First, the kind of each step is clear from its shape; the reading step and the deciding step are in different boxes. Second, the order of the steps is fixed by the arrows; the reader does not have to guess which step comes first. Third, the paths leaving a decision box can be followed separately; on each run only one of these paths is used.

4.2.1.2 Why draw a diagram?

A flowchart is independent of any programming language. The same diagram can be written in Java, Python, or C. That is why the logic of an algorithm can be discussed before it is decided which language it will be coded in. In a team, people who are not programmers, for example a laboratory manager or a business-process manager, can also read the diagram and ask “what happens in this case?”

The second benefit of drawing a diagram is that errors are seen early. If only one arrow leaves a decision box, nobody has thought about what happens in the other case. A box that no arrow enters never runs. If nothing changes along the path of a backward arrow, the loop goes on forever. Such gaps hide between the lines of code; in a diagram they are noticed from the shape of the drawing.

The third benefit is about learning. For a beginner, one of the hardest concepts is the order of execution. The computer runs lines one at a time and in sequence. It skips some lines according to a condition and repeats others. In a flowchart this order can be followed with a finger. Throughout this chapter, every diagram will be executed this way, box by box.

4.2.2 Symbols

The symbols of flowcharts are defined by an international standard: ISO 5807. The standard was published in 1985, and most drawing tools in use today follow it. Five basic symbols are enough for a small algorithm. Additional symbols are used when a diagram grows or when external things such as files and reports are to be shown.

4.2.2.1 Basic symbols

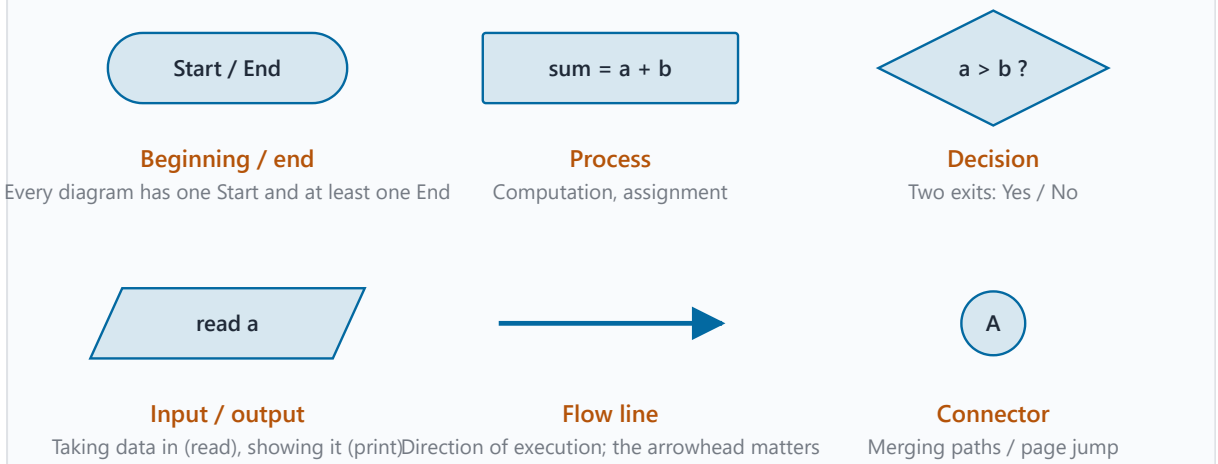


Figure 4.2.2: The basic flowchart symbols. Top row: start/end (oval), process (rectangle), and decision (diamond); bottom row: input/output (parallelogram), flow line (arrow), and connector (small circle).

Figure 4.2.2 shows the basic symbols together. The job of each is explained below.

Definition 4.2.2

The **terminal symbol** (start/end) is an oval or a rectangle with rounded ends. It shows where the algorithm begins and where it ends. Every diagram has a single Start and at least one End.

No arrow enters the Start symbol; only an arrow leaves it. An arrow enters the End symbol; none leaves it. A diagram may have more than one End. For example, in the number-guessing game the game ends immediately on a correct guess; in that case drawing a separate End after the text "You got it" simplifies the diagram. Even so, in most cases merging the paths and leading them to a single End makes reading easier.

Definition 4.2.3

The **process symbol** is a rectangle. It shows a computation or an **assignment**. An assignment gives a variable a new value: the expression `sum = a + b` computes the value `a + b` on the right and writes the result into the variable named `sum`.

The word **variable** appears here for the first time. A variable can be thought of as a box in memory that has a name. At any moment the box holds a single value. When a new value is written, the old one is erased. The equals sign in an assignment is different from equality in mathematics. `i = i + 1` is

meaningless in mathematics, because no number can equal one more than itself. In a flowchart and in a program, however, this statement means "add 1 to the current value of i and write the result back into i." If i was 4 before, it is 5 after this box runs.

Definition 4.2.4

The **decision symbol** is a diamond. A question whose answer is "yes" or "no," that is, a **condition**, is written inside it: `a > b ?`. One arrow enters the decision symbol and two leave it; the outgoing arrows are labeled Yes and No.

A condition is a comparison. In flowcharts and in Java there are six comparison operators. These are `<` (less than), `<=` (less than or equal), `>` (greater than), `>=` (greater than or equal), `==` (equal), and `!=` (not equal). Two equals signs are used for the equality question, because the single equals sign is reserved for assignment. Some books also use the signs ≤, ≥, and ≠; the meaning is the same.

Definition 4.2.5

The **input/output symbol** is a parallelogram. It shows the algorithm taking data from outside ("read a") or giving a result to the outside ("print sum"). Data taken from outside is called **input**, and the result given to the outside is called **output**.

The word "read" covers data coming from the keyboard, from a file, or from a sensor. A flowchart is not concerned with where the data comes from; it only says that at this point a value enters the algorithm. Likewise "print" covers writing to the screen, to a printer, or to a file.

Definition 4.2.6

The **flow line**, or simply **arrow**, joins two symbols and shows the direction of execution. The **connector** is a small circle; it shows the point where several paths meet, or that the diagram continues elsewhere.

The arrowhead matters. A line without a head says that two boxes are connected but not which runs first. That is why every flow line has an arrowhead at its end. The connector marks the place where two paths leaving a decision meet again, as in Figure 4.2.1. A connector with a letter inside is paired with another connector carrying the same letter. This way there is no need to draw a long arrow from one corner of the page to the other.

Symbol	Shape	Job	Example
	Oval	Beginning or end	Start

Symbol	Shape	Job	Example
	Rectangle	Process (computation, assignment)	`sum = a + b`
	Diamond	Decision (condition)	`a > b ?`
	Parallelogram	Input / output	read a
	Arrow	Direction of execution	go to the next step

Table 4.2.1: The basic symbols, their jobs, and sample expressions that can be written inside them.

4.2.2.2 What goes inside a symbol?

Because the shape of a symbol already tells the kind of step, only a short expression of that step is written inside. A decision symbol gets a *question*: `a > b ?`. A process symbol gets a *command*: `sum = a + b`. An input/output symbol gets what is read or printed: "read a," "print sum." This distinction lets the person reading a diagram know what will happen at a step even before reading the inside of the box.

The question form has one more rule: its answer must have two options. The question "which is larger, a or b?" has the answer "a" or "b," but leaves the case of equality open. The question `a > b ?`, on the other hand, always has the answer yes or no: if a = b, the answer is no. That is why a comparison is always written in a decision box.

Example 4.2.1

An algorithm computes a student's average from the midterm and final exam grades and prints the pass/fail result. Its steps are: get the midterm grade; get the final grade; compute the average as midterm × 0.4 + final × 0.6; check whether the average is 50 or above; print the result on the screen. Which symbol does each step get, and what is written inside?

Solution. "Get the midterm grade" is an input: a parallelogram, with "read midterm" inside. The final grade likewise goes in a parallelogram. The average computation is an assignment: a rectangle, with `avg = midterm × 0.4 + final × 0.6` inside. "Is it 50 or above" is a question: a diamond, with `avg >= 50 ?` inside. Printing the result is an output: one parallelogram on each of the Yes and No branches, with "print Pass" and "print Fail" inside. Start is added at the very beginning and End at the very end. If the midterm is 40 and the final is 60, the average is *40 × 0.4 + 60 × 0.6 = 16 + 36 = 52*; since 52 ≥ 50, the Yes path is followed and "Pass" is printed.

Common mistake

Writing a computation in a parallelogram. `sum = a + b` has nothing to do with the screen or the keyboard; it computes a value in memory. It therefore belongs in the process symbol, the rectangle. The parallelogram is only for exchanging data with the outside world. A similar confusion appears when a command is written in a decision box: the sentence “compare a with b and print the larger” is not a question, and it does two jobs at once. The right way is to write `a > b ?` in the decision box and move the printing to parallelograms on the branches.

4.2.2.3 Additional symbols

The five basic symbols are enough for small algorithms. In some cases the additional symbols of ISO 5807 are used. A large diagram may be split across several pages, a separately drawn subprocess may be called, or a file or report may need to be shown. Table 4.2.2 summarizes the six met most often.

Symbol	Name	Job	Example
	Off-page connector	The diagram continues on another page; the page and point are written inside.	2-A
	Predefined process	Calls a separately drawn subprocess.	Compute average
	Preparation	Gives the start and limit of a loop counter in a single symbol.	i = 1 → n
	Document	A printed output or report.	Grade sheet
	Data store	Data kept on disk or in a database.	Student records
	Annotation	Attaches a note to a symbol with a dashed line; takes no part in the flow.	grade 0–100

Table 4.2.2: Additional symbols.

The off-page connector is a relative of the on-page connector. The label “2-A” inside it means “continues at point A on the second page.” The predefined process symbol is a rectangle with two vertical lines on its sides. This symbol is the tool for splitting large diagrams into parts: a subprocess is drawn once and called with this symbol wherever it is needed in the main diagram. The preparation symbol is a hexagon and gathers the start value, limit, and step of a counting loop in one box. The annotation symbol takes no part in the flow; it is joined to a box by a dashed line and gives information to the reader. For example, the annotation “grade 0–100” reminds the reader what range the input must be in.

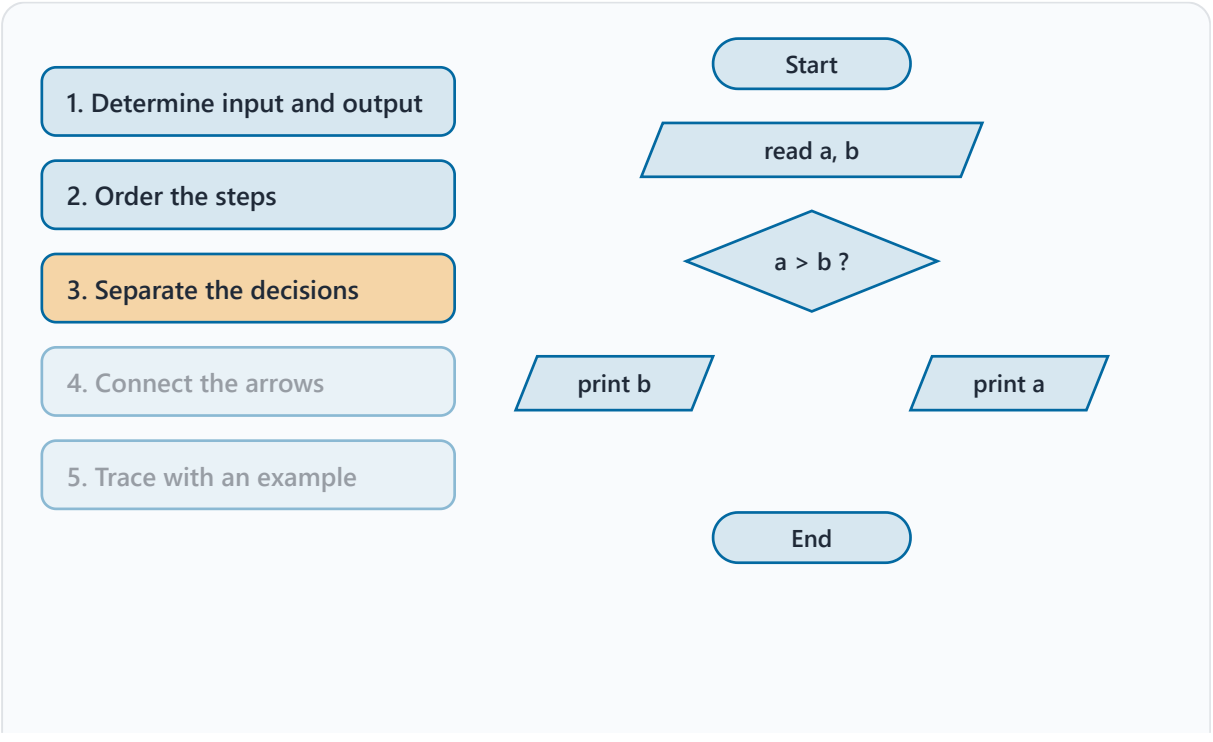
4.2.3 Diagram design

Knowing the symbols is not enough to draw a diagram. The path from a problem to a diagram has definite steps. When these steps are followed, no case is left out, and the result is a drawing that is easy to read.

4.2.3.1 A diagram step by step: the larger of two numbers

The problem is short: two numbers are given and the larger one is to be printed. The diagram is built in five steps.

1. **Determine the input and output.** What the algorithm takes from outside and what it gives to the outside is written first. The input is two numbers: a and b. The output is the larger number. Both are shown with parallelograms: “read a, b” and “print the larger.”
2. **Order the steps.** Start and End are added. The steps are arranged from top to bottom: reading first, printing last. A value that has not been read cannot be compared; that is why the reading box is always above the decision.
3. **Separate the decisions.** Which number is printed depends on a question: is a greater than b? The decision box `a > b ?` goes in between. The single “print the larger” box now splits into two boxes: “print a” and “print b.”
4. **Connect the arrows.** Each box is joined to the next with an arrow. The two arrows leaving the decision are labeled Yes and No. The two branches merge at a connector below and go to End.
5. **Trace with an example.** The diagram is executed by hand with an input. For a = 4, b = 9, *4 > 9* is false; the No path is followed, the “print b” box runs, and 9 is printed. The result is correct.



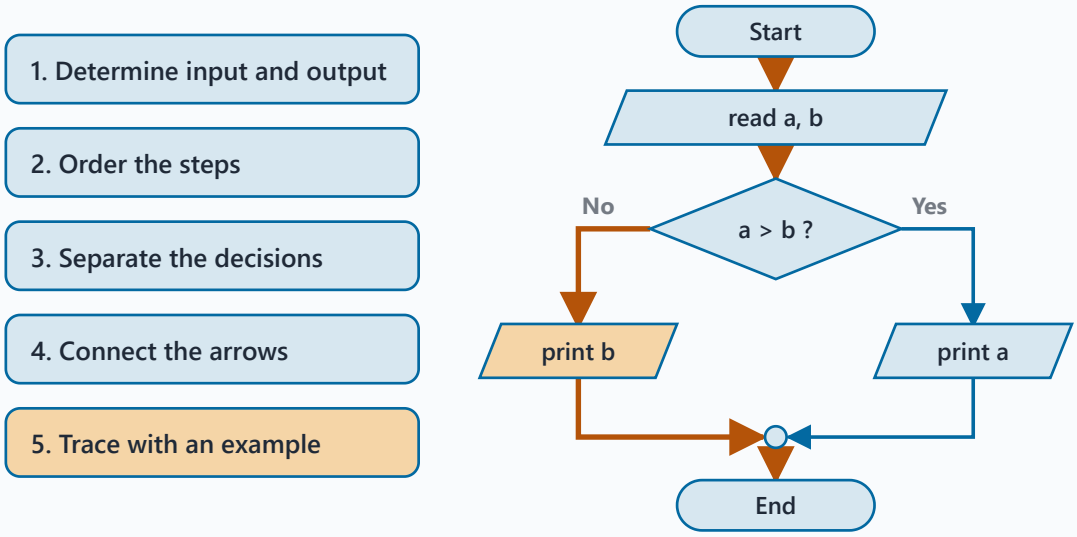


Figure 4.2.3: Building the diagram that finds the larger of two numbers. (a) Step 3: when the decision is added, the output splits into two boxes; the arrows are not there yet. (b) Step 5: the arrows are connected, and the path followed for a = 4, b = 9 is shown in color: the No path, "print b," and 9.

The fifth step is often skipped, but it is the most valuable one. During a hand trace, trying a single input is not enough. The Yes path must also be tried with a = 9, b = 4; in that case 9 is printed. And what if a = b = 5? Since *5 > 5* is false, the No path is followed and b, that is 5, is printed. When the two numbers are equal it does not matter which one is printed; the diagram gives the correct result in this case too. This small experiment shows that the result would not change even if `>=` were written in the decision box instead of `>`.

4.2.3.2 Readability rules

The same algorithm can be drawn well or badly. Figure 4.2.4 shows the same two-number problem in two ways. In the drawing on the left, the arrows cross each other, the exits of the decision box are unlabeled, and three separate jobs, such as "print b, end, a=b," are crammed into one box. The drawing on the right flows from top to bottom, the exits of the decision are labeled, and each box contains a single job.

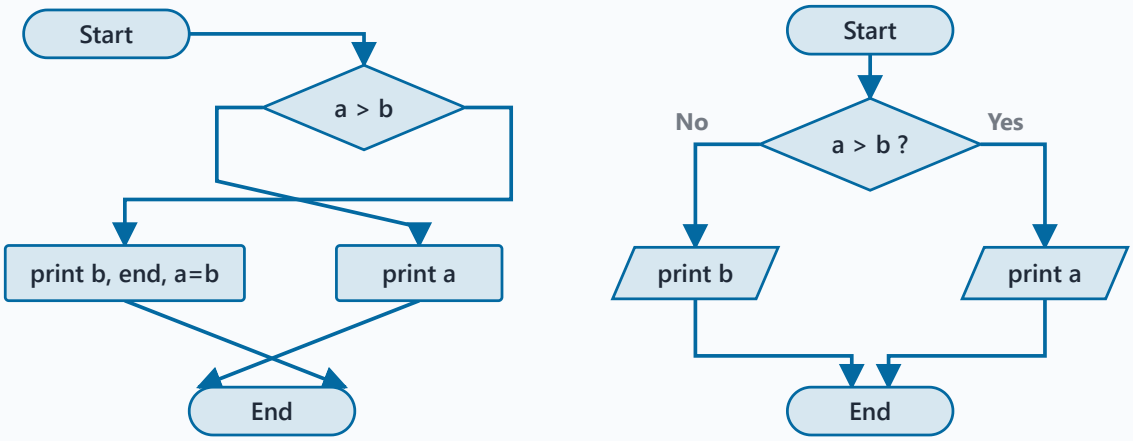


Figure 4.2.4: An unreadable and a readable diagram. On the left, crossing arrows, an unlabeled decision, and three operations in one box; on the right, top-to-bottom flow, a decision labeled Yes/No, and one operation per box.

The rules fall under three headings:

- **Direction of flow.** The flow runs from top to bottom and from left to right as far as possible. The reader's eye moves in this direction; arrows going the other way are used only in loops, to show the return. Crossing arrows are kept to a minimum; if crossing is unavoidable, a connector is used.
- **Decision labels.** Every arrow leaving a decision carries a meaningful label such as Yes/No (or True/False). In an unlabeled decision, which path is chosen in which case is left to guesswork.
- **One job per box.** A single basic operation is written in a symbol. If three operations are crammed into one box, it becomes unclear which is done first, and if the box contains a decision, that decision is overlooked. If a diagram grows and no longer fits on the page, it is split into subprocesses.

Common mistake

Drawing only one arrow out of a decision. The student draws the case in mind, for example “if the grade is above 60, print Pass,” and forgets the other case. When the grade is 45, the diagram does not say where execution goes; the flow breaks. Two arrows always leave a decision box. If there is no job to do in the other case, the No arrow goes directly to the merge point.

4.2.3.3 Flowchart and pseudocode

The second common way of describing an algorithm is pseudocode.

Definition 4.2.7

Pseudocode is a way of writing the steps of an algorithm in short sentences that resemble a programming language but are not bound by its rules. Lines are read in order; lines inside a decision or loop are shown by indentation.

Figure 4.2.5 shows the algorithm that finds the absolute value of a number in two forms. If the number is negative, its sign is changed; otherwise it stays as it is.

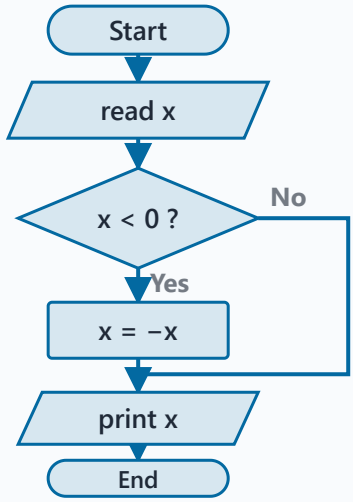


Figure 4.2.5: Flowchart of the absolute-value algorithm: x is read, if x < 0 then x = −x is done, then x is printed.

```
read x
if x < 0 then
    x = -x
print x
```

The two notations describe the same algorithm and either can be translated into the other line by line. “read x” corresponds to the parallelogram, “if x < 0 then” to the decision box, the indented line `x = -x` to the rectangle on the Yes branch of the decision, and “print x” to the last parallelogram. The No branch of the decision has no box; the arrow goes straight to the printing box. In pseudocode this case is shown by a single indented line under the “if” line.

	The flowchart	Pseudocode
Shape	Symbols and arrows	Short indented sentences
Strength	Branching and looping are visible at a glance	Quick to write, very close to code
Weakness	Does not fit on a page when it grows; hard to change	The structure of the flow depends on indentation

	The flowchart	Pseudocode
Best suited for	Learning, presentations, business processes	Long algorithms, right before coding

Table 4.2.3: Comparison of the flowchart and pseudocode.

Pseudocode has no standard. Some books write “if … then,” others “when … do”; some use `=` for assignment, others `←`. What matters is that each step is clear. In this chapter, pseudocode is written with English words and Java-like symbols: assignment `=`, equality question `==`, remainder `%`.

4.2.3.4 The three structures of structured programming

In a flowchart, arrows can be drawn anywhere. An arrow jumping to a random box in the middle of the diagram is technically possible. In the 1960s, as programs grew, this freedom caused serious problems. The arrows tangled, and it was impossible to follow by how many different paths a box could be reached. Such drawings and programs were called **spaghetti code**, after a tangle of pasta.

The solution is to use only three patterns instead of drawing arrows at random. Figure 4.2.6 shows these three patterns.

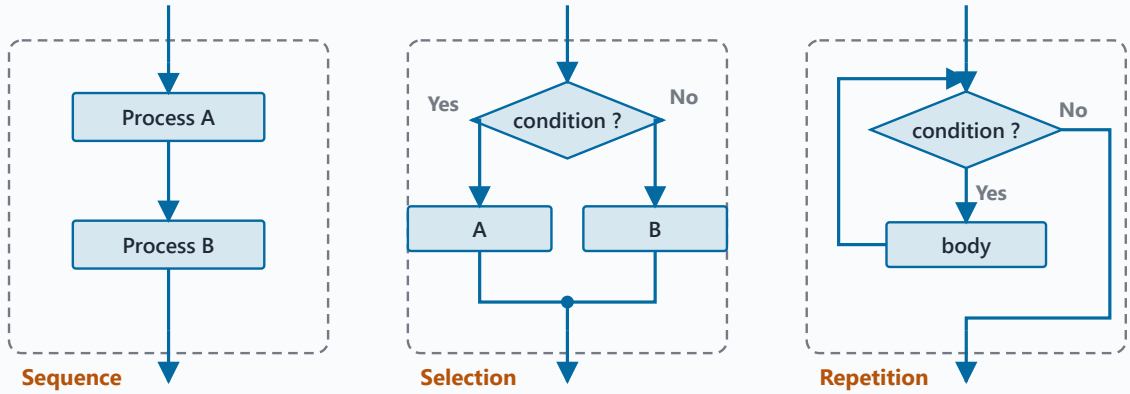


Figure 4.2.6: The three structures of structured programming. Sequence: process A, then process B. Selection: A or B according to the condition. Repetition: the body runs again as long as the condition is true. Each structure has a single entry and a single exit.

Definition 4.2.8

Structured programming is the approach of building algorithms from only three control structures. The **sequence** structure: steps run one after another. The **selection**, or decision, structure: one of two paths runs according to a condition. The **repetition**, or loop, structure: a group of steps runs again as long as a condition is true.

Theorem 4.2.1 (Böhm–Jacopini)

Every algorithm can be written with only the structures of sequence, selection, and repetition. Corrado Böhm and Giuseppe Jacopini published this result in 1966.

What the theorem says is strong: arrows that jump at random are never *needed*. Every diagram drawn in spaghetti form has a structured equivalent that does the same job. If necessary, one or two extra variables (for example a flag) are used.

The common property of the three structures is the **single entry, single exit** rule. One arrow enters each structure from above and one arrow leaves from below. Thanks to this, a structure can be put in place of any box inside another structure. For example, a whole decision structure can be placed instead of the rectangle in the body of a loop. These are called **nested** structures. Large algorithms are built this way, by nesting small structures. Any part of the diagram can be understood and tested separately through its own entry and exit arrows.

Programming languages adopted this idea too. In older languages the `goto` statement, which jumps to any line of the program, was common. In Java, `goto` is a reserved word, that is, it cannot be used as a variable name, but it does nothing in the language. In Java, the flow of execution is built only from lines in sequence, from `if` / `else`, and from the `while`, `do`, and `for` loops.

4.2.3.5 Drawing tools

No special tool is required to draw a flowchart. The choice depends on what the diagram will be used for:

- **Pen and paper:** The fastest way for a first draft; erasing and redrawing is easy. In exams, too, diagrams are drawn by hand.
- **Word and PowerPoint:** All the basic symbols are in the Flowchart group of the Insert → Shapes menu. Because connector arrows stick to the boxes, the arrow moves with a box when it is moved.
- **draw.io (diagrams.net):** It is free and runs in the browser. It offers ready-made flowchart shapes; the drawing can be exported as PNG, SVG, or PDF.
- **Microsoft Visio:** A commercial tool used for corporate process and workflow diagrams.
- **Flowgorithm:** It does not merely draw the diagram but runs it. It shows the values of variables step by step and translates the diagram into code in many languages, including Java.
- **Mermaid:** The diagram is defined with text and the tool draws it. It is often used in documents and code repositories.

Whatever the tool, the meaning of the symbols and the readability rules are the same. The course's Flowchart Runner simulator, like Flowgorithm, executes a diagram step by step; most of the examples in these notes can be tried there as well.

4.2.4 Sequential flow

The simplest of the three structures is sequence. All the control structures are built on this foundation: any section without a decision or loop is sequential flow in itself.

4.2.4.1 The sequence structure

Definition 4.2.9

Sequential flow is flow in which the steps run in the order they are written, each after the previous one has finished, and exactly once. It contains no branching or repetition; it has a single entry and a single exit.

Figure 4.2.7 shows the diagram that reads two numbers and prints their sum. All the arrows in the diagram point downward; no box has more than one arrow leaving it.

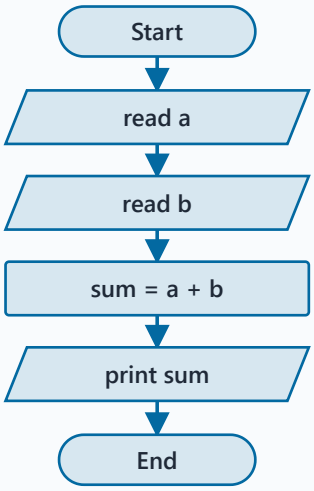


Figure 4.2.7: Sequential flow: Start, read a, read b, sum = a + b, print sum, End.

```
Start
  read a
  read b
  sum = a + b
  print sum
End
```

In the pseudocode, the four lines between Start and End are indented; each line corresponds to one box in the diagram.

4.2.4.2 Executing sequential flow step by step

To **execute** a diagram is to do by hand the work the computer would do. We start at the Start box. At each step only one box runs. After a box runs, the new values of the variables and the output, if any, are noted, and then we move to the box the arrow points to. If these notes are collected in a table, the result is a **trace table**. Table 4.2.4 is the trace table of the execution with input a = 4, b = 7.

Step	Box executed	a	b	sum	Output
1	Start	—	—	—	
2	read a	4	—	—	
3	read b	4	7	—	
4	sum = a + b	4	7	11	
5	print sum	4	7	11	11
6	End	4	7	11	11

Table 4.2.4: Trace table of the sequential flow for a = 4, b = 7. A dash shows that the variable does not yet have a value.

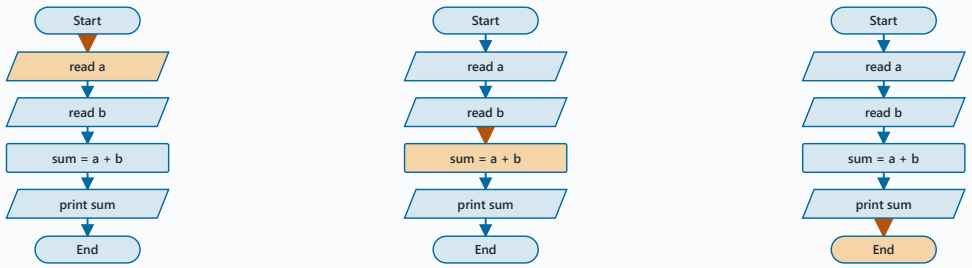


Figure 4.2.8: Three steps of the sequential flow; the box running at that moment is highlighted. (a) Step 2: a = 4 is read. (b) Step 4: sum = 4 + 7 = 11. (c) Step 6: End is reached.

The table shows two properties of sequential flow clearly. First, each of the six boxes runs exactly once; there is no repeated or skipped row in the table. Second, once a variable receives a value, it keeps it unless it is changed: a becomes 4 at step 2 and stays 4 to the end. When reading the table, each row describes the state “right after this box has run.”

The Java counterpart of the diagram follows the same order:

```java
import java.util.Scanner;

public class Sum {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int a = in.nextInt();             // read a
        int b = in.nextInt();             // read b
        int sum = a + b;                  // sum = a + b
        System.out.println(sum);          // print sum
    }
}
```

The structure of the program can be summed up in a few words. The line `public class Sum` gives the program's name. The `main` method is where the program starts; the counterpart of the Start box in the diagram is its opening brace, and the counterpart of End is the closing brace. `Scanner` is the ready-made class Java provides for reading data from the keyboard; `in.nextInt()` reads an integer from the keyboard. That is why each “read” parallelogram becomes a call to `nextInt()`. The assignment rectangle corresponds exactly to the line `int sum = a + b;`, and the “print” parallelogram to the line `System.out.println(sum);`. At the ends of the lines there are comments beginning with `//`. The compiler ignores them; here they remind us of each line's counterpart in the diagram.

When the program is run with input 4 and 7, it prints 11. With input −3 and 10, it prints 7. In a sequential program, even when the inputs change, the lines that run and their order never change; only the values do.

4.2.4.3 Why order matters

In sequential flow, the positions of the boxes are not arbitrary. If the sum box were moved above the reading boxes, a and b would not yet have values and the sum could not be computed. This mistake is easy to notice. More insidious bugs arise from swapping two assignments.

Example 4.2.2

The values of two variables are to be exchanged; this operation is called a **swap**. If a = 4, b = 7, then after the operation a = 7, b = 4 must hold. A student puts two boxes in the diagram: first `a = b`, then `b = a`. What is the result? How is the correct diagram built?

Solution. The proposed diagram is executed step by step. When the `a = b` box runs, a takes the value of b: a = 7, b = 7. a's old value, 4, is now nowhere. When the `b = a` box runs, b takes a's current value: b = 7. The result is a = 7, b = 7; the swap has failed.

The problem is that the first assignment erases a's old value. The solution is to store that value first in a third variable. Such a briefly used variable is called a **temporary variable**. The correct order is three boxes: `temp = a`, `a = b`, `b = temp`. Execution: temp = 4; a = 7; b = 4.

The result is a = 7, b = 4.

```
import java.util.Scanner;

public class Swap {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int a = in.nextInt();
        int b = in.nextInt();
        int temp = a;                    // a's value is saved
        a = b;
        b = temp;
        System.out.println(a + " " + b);
    }
}
```

When the program is run with input 4 7, it prints `7 4`. The example shows the basic rule of sequential flow: each box works with the values the previous one left behind. An assignment reads the current value of the variable on its right side; not its old or future value.

Common mistake

Reading an assignment like an equality in mathematics. `a = b` does not say “a and b are equal”; it says “copy the current value of b into a.” After the copy, if b changes, a does not; if a is changed, b is not. Two variables carry the same value only at the moment of the assignment.

4.2.5 Conditional flow

A sequential algorithm does the same steps on every input. In most real problems, however, the work to be done depends on the situation: if the grade is below 60 the student fails, if the code is wrong the door does not open, if the temperature is below zero a warning is given. Such choices are shown with the decision symbol.

4.2.5.1 The two-way decision

Definition 4.2.10

Conditional flow is flow in which one of two paths is followed according to the result of the condition asked in a decision symbol. If the condition is true, the Yes path runs; if false, the No path. These paths are called **branches**, and the splitting of the path in two is called **branching**. The branches merge again at a connector.

Figure 4.2.9 shows the diagram that prints “Pass” or “Fail” according to an exam grade. The passing mark is 60; 60 and above passes.

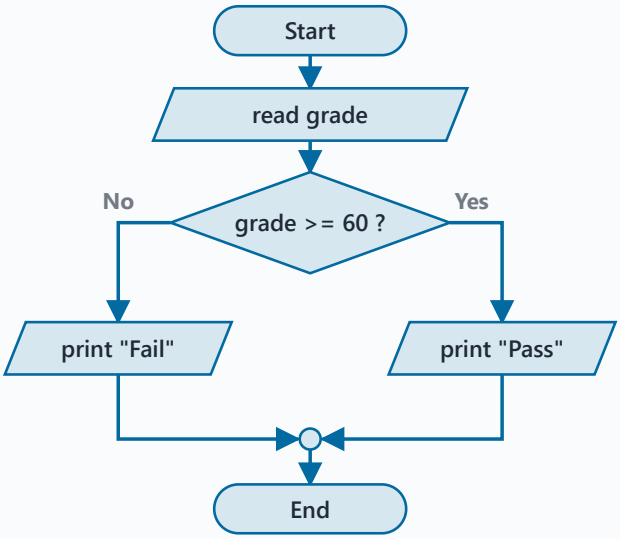


Figure 4.2.9: Conditional flow: if the grade is 60 or above, “Pass” is printed; otherwise “Fail.” The paths merge at the connector below.

```
Start
  read grade
  grade >= 60 ?
    Yes → print "Pass"
    No → print "Fail"
End
```

Three points in the diagram are especially important. Two paths leave the decision symbol, but on each run only one of them is followed. The “Pass” and “Fail” boxes never run together in the same run. The paths merge again at the connector below (●); this point is the single exit of the decision structure. Finally, the boundary value of the condition deserves attention. Because the passing mark is “60 and above,” the condition is written `grade >= 60`. Had `grade > 60` been written, a student who scored exactly 60 would fail.

4.2.5.2 Executing the decision step by step

To fully understand a conditional diagram, it must be executed at least twice. One of the inputs must run the Yes path and the other the No path. Table 4.2.5 summarizes the diagram on three inputs: grade = 72, grade = 45, and the boundary value grade = 60.

grade	Decision	Arrow followed	Box executed	Box that never runs
72	72 >= 60 → true	Yes	print "Pass"	No branch
45	45 >= 60 → false	No	print "Fail"	Yes branch
60	60 >= 60 → true	Yes	print "Pass"	No branch

Table 4.2.5: Three runs of the Pass/Fail diagram. In each run one of the boxes never runs.

In the first run the user enters 72 and grade = 72. The decision box asks *72 >= 60*; the answer is true. Flow continues along the Yes arrow and “Pass” is printed. The “Fail” box never runs in this run. The path rejoins the main flow at the connector and End is reached. In the second run grade = 45. *45 >= 60* is false; the No path is followed, “Fail” is printed, and this time the “Pass” box does not run. Together, the two runs have executed every box in the diagram at least once.

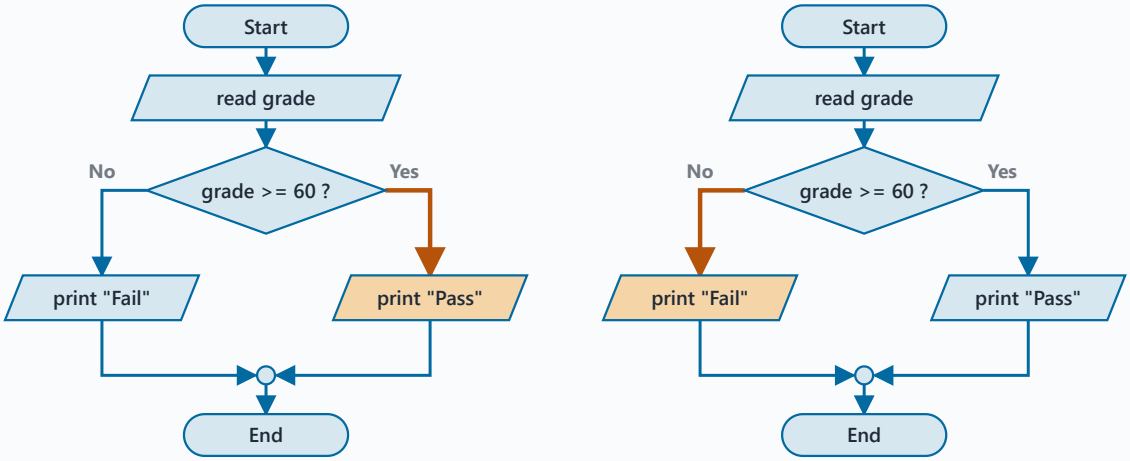


Figure 4.2.10: Two runs of the same diagram. (a) grade = 72: the “print Pass” box on the Yes branch runs. (b) grade = 45: the “print Fail” box on the No branch runs.

The third row is the boundary value. When 60 is entered, *60 >= 60* is true and “Pass” is printed. This row immediately exposes a diagram in which `>` was mistakenly written instead of `>=` in the condition: that diagram would print “Fail” for 60. 72 and 45 cannot catch this error, because the two diagrams give the same result on these inputs.

The Java counterpart of the diagram is the `if` / `else` statement:

```
import java.util.Scanner;

public class PassFail {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int grade = in.nextInt();          // read grade
        if (grade >= 60) {                 // grade >= 60 ?
            System.out.println("Pass");   // Yes branch
        } else {
            System.out.println("Fail");   // No branch
        }
    }
}
```

After the word `if`, the condition is written in parentheses. If the condition is true, the inside of the first pair of braces runs; if false, the inside of the braces after the word `else` runs. The decision symbol corresponds to the condition in parentheses, the Yes branch to the first block, and the No branch to the `else` block. The last closing brace also stands in for the connector: after the two blocks end, execution continues from the same point. The program prints “Pass” for 72, 60, and 100, and “Fail” for 45, 59, and 0.

4.2.5.3 The one-way decision

In some decisions, only one of the branches has work to do. The absolute-value diagram in Figure 4.2.5 is an example: if the number is negative its sign is changed; if it is not negative, nothing is done. Two arrows still leave the decision here, but the No arrow goes straight to the merge point without passing through any box. In Java this structure is written with an `if` that has no `else` part:

```
import java.util.Scanner;

public class AbsoluteValue {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int x = in.nextInt();
        if (x < 0) {                       // no box on the No branch: no else needed
            x = -x;
        }
        System.out.println(x);
    }
}
```

The program prints 5 for −5, 8 for 8, and 0 for 0. The three inputs are chosen deliberately: −5 tries the Yes branch, 8 the No branch, and 0 the boundary. Since *0 < 0* is false, 0 stays as it is; its absolute value is 0 anyway.

Another common example of a decision structure is the odd/even test. Whether a number is even is found by looking at the remainder of its division by 2. The **remainder** operation is written `%` in Java: `7 % 2` is 1 and `12 % 2` is 0. The decision box gets `x % 2 == 0 ?`; the Yes branch prints “even” and the No branch prints “odd.”

```
import java.util.Scanner;

public class EvenOdd {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int x = in.nextInt();
        if (x % 2 == 0) {                // x % 2 == 0 ?
            System.out.println("even");
        } else {
            System.out.println("odd");
        }
    }
}
```

Common mistake

Testing for an odd number with the condition `x % 2 == 1 ?`. This condition works for positive numbers, but in Java the sign of the remainder follows the sign of the dividend: `-3 % 2` is −1, not 1. So for −3, `x % 2 == 1` comes out false and an odd number is printed as “even.” The question `x % 2 == 0 ?`, on the other hand, gives the right answer for −3 too: `-3 % 2 == 0` is false, and “odd” is printed. When the remainder is compared with zero, the sign problem disappears.

4.2.5.4 The largest of three numbers

Finding the larger of two numbers needs a single decision. With three numbers the decisions multiply, and how the diagram is built starts to matter. A widely used method is the “candidate” idea. The first number is taken as the largest for now. Then the other numbers are compared with this candidate one by one; if a larger one turns up, the candidate changes.

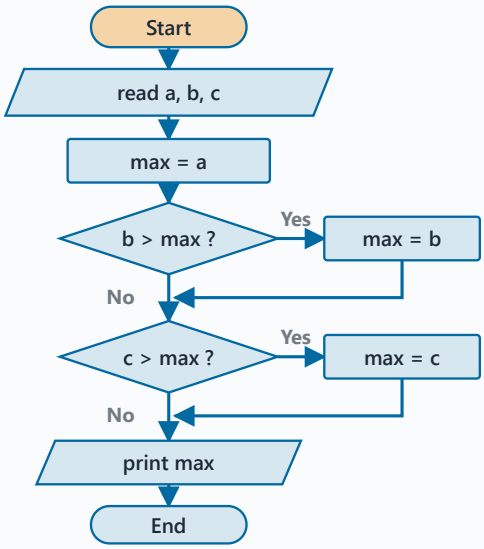


Figure 4.2.11: The largest of three numbers. We start with max = a; if b is larger, max = b; then if c is larger, max = c.

```
Start
  read a, b, c
  max = a
  b > max ?  Yes → max = b
  c > max ?  Yes → max = c
  print max
End
```

Figure 4.2.11 is the diagram of the method. Two decisions come one after the other, and each has a box only on its Yes branch; the No branches pass straight to the next step. That is, the diagram consists of two one-way decisions placed one after the other inside a sequence. It is a nice example of structures growing by being attached to one another.

Example 4.2.3

The diagram will be executed first for (a, b, c) = (4, 9, 7), then for (3, 2, 8). Which path is followed at each decision, and how does max change?

Solution. In the first run the three numbers are read and max = 4. At the first decision *9 > 4* is true; max = 9. At the second decision *7 > 9* is false; max does not change and 9 is printed.

Box executed	Decision	max
max = a	—	4
b > max ?	9 > 4 → Yes	4
max = b	—	9

Box executed	Decision	max
c > max ?	7 > 9 → No	9
print max	output: 9	9

Trace for (4, 9, 7)

In the second run we start with max = 3. *2 > 3* is false; the Yes box of the first decision does not run. *8 > 3* is true; max = 8 and 8 is printed.

Box executed	Decision	max
max = a	—	3
b > max ?	2 > 3 → No	3
c > max ?	8 > 3 → Yes	3
max = c	—	8
print max	output: 8	8

Trace for (3, 2, 8)

Together, the two runs have tried both branches of both decisions. Each decision is asked only once. Even if the quantity of numbers grows, the method does not change: for ten numbers, nine decisions are lined up one after another.

```
import java.util.Scanner;

public class Largest {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int a = in.nextInt(), b = in.nextInt(), c = in.nextInt();
        int max = a;                        // first candidate
        if (b > max) {
            max = b;
        }
        if (c > max) {
            max = c;
        }
        System.out.println(max);
    }
}
```

The program prints 9 for (4, 9, 7) and 8 for (3, 2, 8). On the input (9, 4, 7), where the largest number comes first, both decisions are No and 9 is printed. If the three numbers are equal, for example (5, 5, 5), the decisions are again No; max stays 5.

Common mistake

Starting the candidate with a fixed value, for example `max = 0`, instead of the first number. As long as the inputs are positive this also gives the right result; the error appears only when all the inputs are negative. For (−2, −7, −1), *−2 > 0*, *−7 > 0*, and *−1 > 0* are all false; the candidate never changes and 0 is printed. Yet 0 is not even one of the inputs. The correct diagram starts with `max = a` and prints −1 on this input.

4.2.5.5 The multi-way decision: a decision chain

A decision box has only two exits. When there are more than two options, the decisions are lined up like a chain: each decision is placed on the No branch of the previous one. This is called a **multi-way decision**.

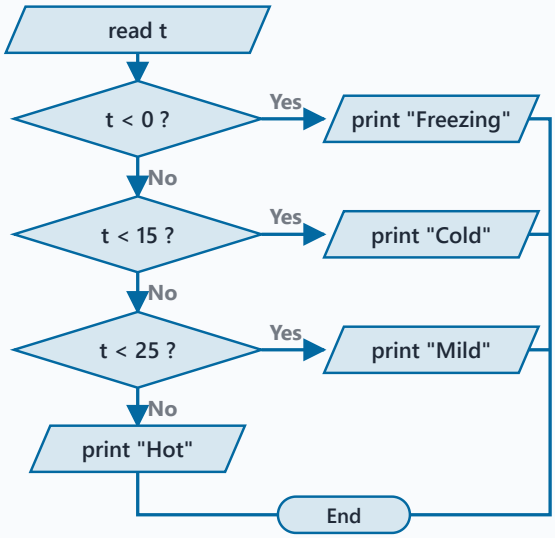


Figure 4.2.12: The decision chain: for temperature t, “Freezing” is printed if t < 0, “Cold” if t < 15, “Mild” if t < 25, and “Hot” otherwise.

Figure 4.2.12 sorts a temperature value into one of four classes. The rule by which the chain works is this: the decisions are asked in order from top to bottom; the branch of *the first true condition* runs, and the decisions after it are never asked. If no condition is true, the bottom No branch runs.

s	Decisions asked	Number of decisions	Output
−5	t < 0 ? Yes	1	Freezing
10	t < 0 ? No · t < 15 ? Yes	2	Cold
15	t < 0 ? No · t < 15 ? No · t < 25 ? Yes	3	Mild
18	t < 0 ? No · t < 15 ? No · t < 25 ? Yes	3	Mild

s	Decisions asked	Number of decisions	Output
30	t < 0 ? No · t < 15 ? No · t < 25 ? No	3	Hot

Table 4.2.6: Executing the decision chain on five inputs. The number of decisions asked depends on the position of the first true condition.

The t = 18 row of Table 4.2.6 reads step by step as follows. The first decision is the question *18 < 0*; it is false and the No branch goes to the second decision. *18 < 15* is also false. The third decision, *18 < 25*, is true; “Mild” is printed. For t = −5, on the other hand, the first decision is true and the other two decisions are never asked.

Each decision in the chain checks only the *upper* bound. When the question t < 15 is asked, it is certain that t is 0 or greater; otherwise execution would never have reached this decision. So there is no need for a two-sided condition such as “0 ≤ t and t < 15.” This rule also shows which class the boundary values fall into: for t = 15, *15 < 15* is false, the decision moves on along the chain, and 15 is “Mild.” t = 0 is “Cold.”

The order of the decisions matters. If the decisions were lined up in reverse order, that is, if `t < 25 ?` were asked first, then for t = −5 *−5 < 25* comes out true and “Mild” is printed; the “Freezing” branch is never reached. The same happens for t = 10. The general rule is short: ask *the narrow condition first, the wide condition later*. If the number of options is n, n − 1 decisions are needed; for four classes there are three decisions.

In Java, a decision chain is written in the form `if … else if … else`. Each `else if` is the No branch of the previous condition; the last `else` is the case where no condition held:

```
import java.util.Scanner;

public class Temperature {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int t = in.nextInt();
        if (t < 0) {
            System.out.println("Freezing");
        } else if (t < 15) {
            System.out.println("Cold");
        } else if (t < 25) {
            System.out.println("Mild");
        } else {
            System.out.println("Hot");
        }
    }
}
```

The program prints “Freezing” for −5, “Cold” for 0 and 10, “Mild” for 15 and 18, and “Hot” for 25 and 30. These seven inputs try the four exits of the chain and every boundary value.

4.2.5.6 Compound conditions: AND and OR

Some decisions require two conditions to be evaluated together. A car-rental company requires the customer both to be at least 18 years old and to hold a driver's license. A calendar app prints "Weekend" if the day is Saturday *or* Sunday. Expressions like these, which combine two conditions, are called **compound conditions**.

Definition 4.2.11

A compound condition joined with **AND** is true if both conditions are true; if even one is false, it is false. A compound condition joined with **OR** is true if at least one of the conditions is true; if both are false, it is false. In Java, AND is written with the operator `&&` and OR with `||`.

A compound condition can be written in a single decision box: `age >= 18 AND has license ?`. If detail is needed in the diagram, it is expanded into two separate decisions. Figure 4.2.13 shows how the two connectives are drawn with two decisions.

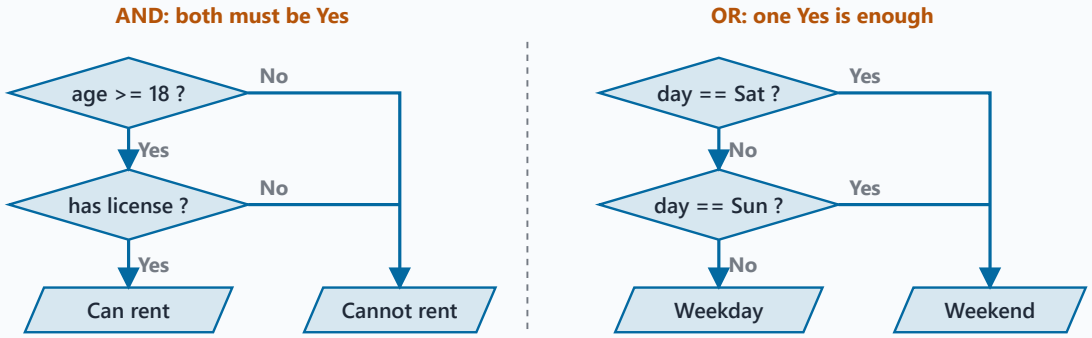


Figure 4.2.13: Drawing a compound condition with two decisions. On the left, AND: both decisions must be Yes. On the right, OR: it is enough if one of the decisions is Yes.

In the AND structure, the second decision is placed on the *Yes* branch of the first. If the first decision is No, the result is already No and the second decision is never asked. In the OR structure, the second decision is on the *No* branch of the first. If the first decision is Yes, the result is already Yes and there is no need for the second. Table 4.2.7 and Table 4.2.8 show this behavior for all cases.

age	license	age >= 18 ?	has license ?	Result
20	yes	Yes	Yes	Can rent
20	no	Yes	No	Cannot rent
16	yes	No	not asked	Cannot rent
16	no	No	not asked	Cannot rent

Table 4.2.7: The AND decision. When the first decision is No, the second is not asked.

day	day == Sat ?	day == Sun ?	Result
Sat	Yes	not asked	Weekend
Sun	No	Yes	Weekend
Tue	No	No	Weekday

Table 4.2.8: The OR decision. When the first decision is Yes, the second is not asked.

Definition 4.2.12

Short-circuit evaluation means that the second condition is not evaluated at all when the first condition already settles the result of the compound condition. In `A && B`, if A is false, B is not looked at; in `A || B`, if A is true, B is not looked at.

The operators `&&` and `||` in Java work in the same way as the two-decision diagrams in Figure 4.2.13. This behavior not only saves time but also prevents some errors. In the condition `count > 0 && total / count > 50`, if count is 0 the second part is never evaluated; so no division-by-zero error occurs.

```
import java.util.Scanner;

public class Rental {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int age = in.nextInt();
        boolean hasLicense = in.nextBoolean();   // true: has a license
        String day = in.next();                  // Mon, Tue, ..., Sat, Sun
        if (age >= 18 && hasLicense) {
            System.out.println("Can rent");
        } else {
            System.out.println("Cannot rent");
        }
        if (day.equals("Sat") || day.equals("Sun")) {
            System.out.println("Weekend");
        } else {
            System.out.println("Weekday");
        }
    }
}
```

With the input “20 true Sat” the program prints “Can rent” and “Weekend”; with the input “16 true Sun” it prints “Cannot rent” and “Weekend.” To test a compound condition, both exits of each decision are tried. In the AND example, the inputs (20, yes), (20, no), and (16, —) run three different paths. Whether a 16-year-old has a license does not change the result, because the second decision is not asked.

4.2.5.7 Leap year

The classic example showing the importance of order in a decision chain is the leap-year rule. In the Gregorian calendar the rule has three parts: years divisible by 4 are leap years; but those divisible by 100 are not; and those divisible by 400 are leap years again. In a leap year February has 29 days.

The rule has exceptions, and the exceptions have exceptions. When such a rule is turned into a diagram, the narrowest condition goes first. Divisibility by 400 is a special case of divisibility by 100, and divisibility by 100 is a special case of divisibility by 4. That is why the decisions are asked in the order 400, 100, 4.

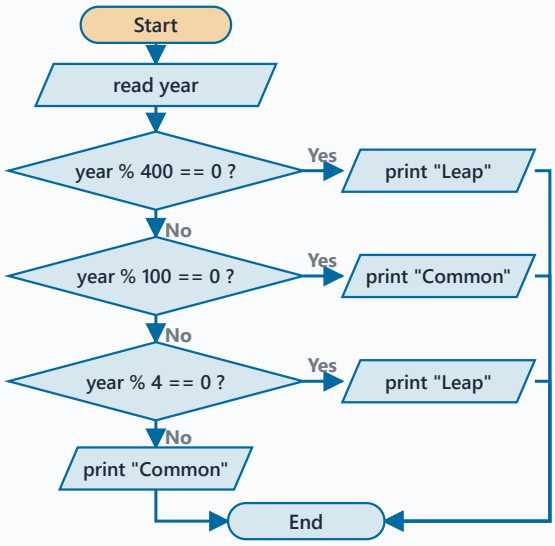


Figure 4.2.14: The leap-year diagram: a chain of three decisions made up of year % 400, year % 100, and year % 4; four exits.

```
Start
  read year
  year % 400 == 0 ?  Yes → print "Leap"
  year % 100 == 0 ?  Yes → print "Common"
  year % 4 == 0 ?    Yes → print "Leap"
  No → print "Common"
End
```

year	year % 400	year % 100	year % 4	Decision asked	Output
2024	24 → No	24 → No	0 → Yes	3	Leap
1900	300 → No	0 → Yes	not asked	2	Common
2000	0 → Yes	not asked	not asked	1	Leap
2023	23 → No	23 → No	3 → No	3	Common

Table 4.2.9: Executing the leap-year diagram on four inputs. Each cell shows the remainder and the result of the decision.

Table 4.2.9 shows the path each of the four inputs takes. For 2024, *2024 % 400 = 24* and *2024 % 100 = 24* are not zero; both decisions are No. Since *2024 % 4 = 0*, the third decision is Yes and “Leap” is printed. For 1900, *1900 % 400 = 300* is not zero but *1900 % 100 = 0*. The second decision is Yes, “Common” is printed, and the third decision is never asked. 1900 is not a leap year even though it is divisible by 4; a diagram that looked only at the question `year % 4 == 0` would be wrong here. For 2000 the first decision is Yes: although 2000 is divisible by 100, it is also divisible by 400, so it is a leap year. 2023 runs the last branch, where all three decisions are No.

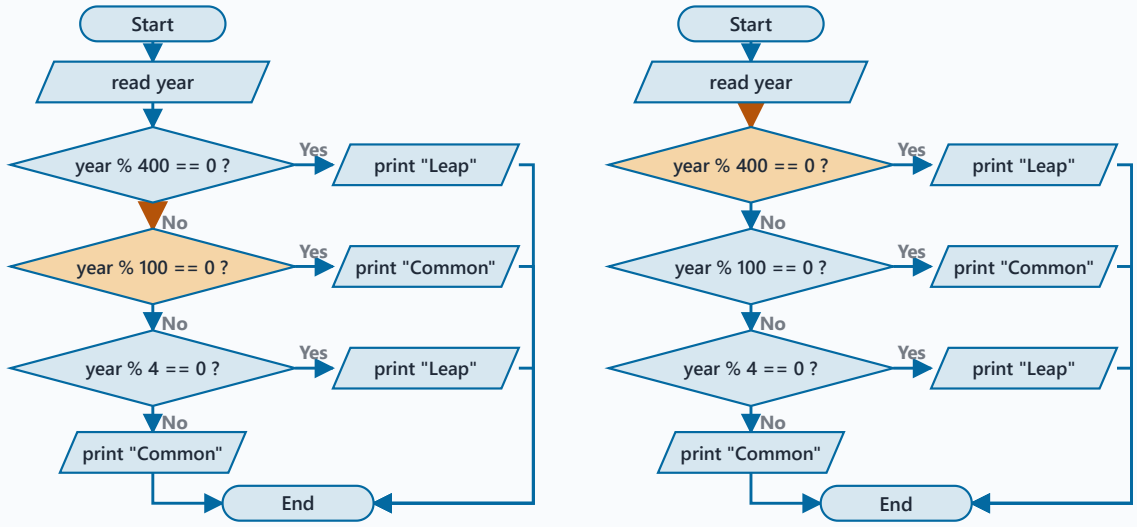


Figure 4.2.15: Two early exits in the leap-year chain. (a) 1900: the second decision (year % 100 == 0) is Yes; the third decision is not asked. (b) 2000: the first decision is Yes; the remaining two decisions are not asked.

The four inputs run each of the diagram's four exits once. This is exactly the goal when testing a decision chain. The Java counterpart is again an `if … else if … else` chain:

```
import java.util.Scanner;

public class LeapYear {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int year = in.nextInt();
        if (year % 400 == 0) {
            System.out.println("Leap");
        } else if (year % 100 == 0) {
            System.out.println("Common");
        } else if (year % 4 == 0) {
            System.out.println("Leap");
        } else {
            System.out.println("Common");
        }
    }
}
```

The program prints “Leap” for 2024 and 2000, and “Common” for 1900 and 2023. The first exception in the near future is the year 2100: 2100 is divisible by 100 but not by 400; the program prints “Common” for this year. 1600 is a leap year because it is divisible by 400.

Common mistake

Starting the chain with the wide condition. If `year % 4 == 0 ?` is asked first and “Leap” is printed directly on its Yes branch, 1900 and 2100 are also counted as leap years. Because the 100 and 400 rules concern only years divisible by 4, their turn never comes. The decisions in the chain must be arranged from the narrow condition to the wide one.

4.2.6 Repetitive flow

What makes a computer powerful is that it can do the same job thousands of times without getting tired. Adding up the grades of 30 students in a class means doing the same two steps 30 times: read the grade, add it to the total. Instead of drawing these steps 30 times in the diagram, we draw them once and make them repeat with an arrow that goes back.

4.2.6.1 The loop structure

Definition 4.2.13

Repetitive flow, or a **loop**, is flow in which a group of steps runs again and again as long as a condition is true. The repeated steps are called the **loop body**, and one run of the body is called an **iteration**. The arrow that returns from the end of the body to the condition forms the loop.

Figure 4.2.16 shows the diagram that prints the numbers from 1 to 5. The diagram has three parts. The `i = 1` box gives the initial value of the counter. The decision box `i <= 5 ?` decides whether the loop continues. The two boxes on the Yes branch, “print i” and `i = i + 1`, are the loop body. The arrow leaving the end of the body turns left and enters the decision box again.

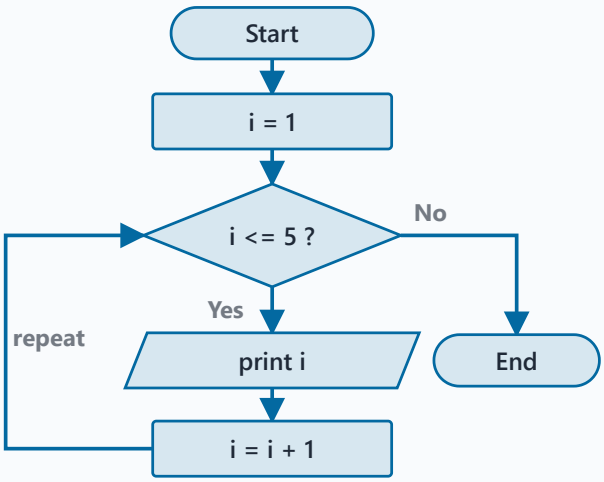


Figure 4.2.16: Repetitive flow: we start with i = 1; while i <= 5, i is printed and increased by 1. The "repeat" arrow on the left forms the loop.

```
Start
  i = 1
  i <= 5 ?
    Yes → print i
            i = i + 1
            back to the decision
    No → End
```

In this diagram the condition is asked *before* the body. If the condition is false on the very first try, the body never runs. Such loops are called **pre-test loops**. When the condition is false, the No branch leaves the loop and goes to End.

For the loop to end some day, there must be an operation on the repeat path that changes the condition. Here this operation is `i = i + 1`. i grows on every iteration and eventually exceeds 5. If this box were forgotten, i would stay 1, *1 <= 5* would come out true every time, and the diagram would print 1 forever. A loop that never ends is called an **infinite loop**.

4.2.6.2 Executing a loop step by step

Because the same boxes run again and again in a loop, each iteration is shown with one row in the trace table. Table 4.2.10 gives all the iterations of the diagram.

Iteration	i (before the decision)	Decision	Printed	i (at the end of the iteration)
1	1	1 <= 5 → true	1	2
2	2	2 <= 5 → true	2	3
3	3	3 <= 5 → true	3	4
4	4	4 <= 5 → true	4	5

Iteration	i (before the decision)	Decision	Printed	i (at the end of the iteration)
5	5	5 <= 5 → true	5	6
—	6	6 <= 5 → false	—	—

Table 4.2.10: Trace table of the loop that prints the numbers 1 to 5. The last row is the final decision, where the loop is exited.

The execution goes as follows. The counter gets its initial value: i = 1. At the decision box *1 <= 5* is asked; it is true and we enter the Yes branch, that is, the loop body. 1 is printed. The `i = i + 1` box makes i 2. The returning arrow takes execution back to the decision box. *2 <= 5* is also true; 2 is printed and i = 3. The same cycle repeats for 3, 4, and 5. After 5 is printed, i = 6. At the decision box *6 <= 5* is asked; this time the answer is false and the No branch leads to End.

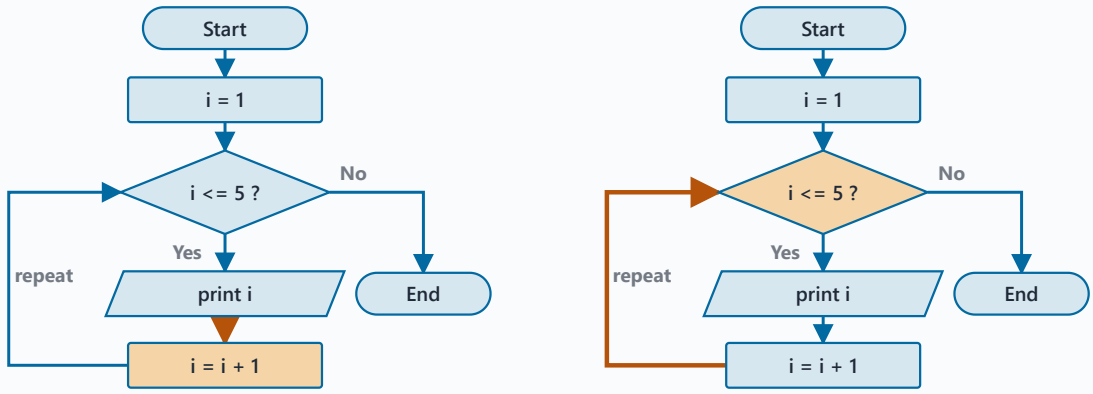


Figure 4.2.17: Two moments of the loop. (a) End of iteration 1: i = 1 + 1 = 2; the returning arrow leads to the decision. (b) The final decision: 6 <= 5 is false; the No branch leaves the loop.

Two counts come out of the table. The body ran 5 times but the condition was asked 6 times. This holds for every pre-test loop: the decision runs one more time than the body, because the last attempt is the false answer that ends the loop. When the loop is exited, the value of i is 6, not 5. The value of the counter after the loop is a question that is often asked and often answered wrongly.

In Java, a pre-test loop is written with `while`. As long as the condition in parentheses is true, the inside of the braces runs again; the closing brace stands in for the returning arrow:

```
public class OneToFive {
    public static void main(String[] args) {
        int i = 1;                      // initial value of the counter
        while (i <= 5) {                // decision: i <= 5 ?
            System.out.print(i + " ");  // print i
            i = i + 1;                  // update the counter
        }                               // returning arrow: back to the decision
        System.out.println();
    }
}
```

The program prints `1 2 3 4 5`. Unlike `println`, `System.out.print` adds no line break; the numbers are therefore printed on the same line with a space between them. The `System.out.println()` line after the loop only ends the line.

4.2.6.3 Counter and accumulator

Most loops contain two kinds of variable. One counts which iteration the loop is on; the other holds the result that builds up from iteration to iteration.

Definition 4.2.14

A **counter** is a variable that tracks how many repetitions have been made; it increases by 1 on every iteration. An **accumulator** is a variable that holds the partial result; on every iteration a new value is added to it (or multiplied into it). The starting value of the accumulator is chosen according to the operation: 0 for a sum, 1 for a product.

The choice of starting value is not arbitrary. For addition we choose 0, because adding 0 to a number does not change it; after the first iteration, the sum is the first number itself. For a product, 1 plays the same role. If a product accumulator is mistakenly started at 0, it is multiplied by 0 on every iteration and the result is always 0. For example, the product of the numbers from 1 to 5 is $1 \times 1 \times 2 \times 3 \times 4 \times 5 = 120$; if the accumulator starts at 0, we get $0 \times 1 \times 2 \times 3 \times 4 \times 5 = 0$.

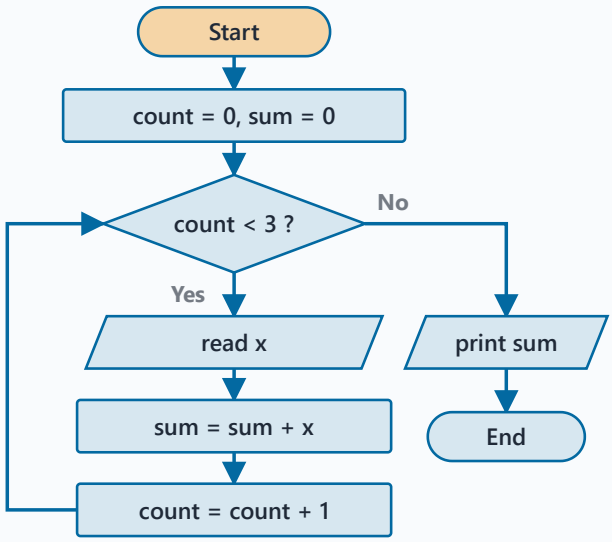


Figure 4.2.18: Counter and accumulator: three numbers are read and summed. While count < 3, x is read, sum = sum + x and count = count + 1 are done.

```
Start
  count = 0, sum = 0
  count < 3 ?
    Yes → read x
            sum = sum + x
            count = count + 1
            back to the decision
    No → print sum
End
```

Figure 4.2.18 is the diagram that reads three numbers and prints their sum. `count` is the counter and `sum` is the accumulator. Both receive the value 0 once, before the loop. Table 4.2.11 gives their values at the end of each iteration when the inputs are 4, 7, and 2.

State	x	count	sum	Next decision
start	—	0	0	0 < 3 → true
Iteration 1	4	1	4	1 < 3 → true
Iteration 2	7	2	11	2 < 3 → true
Iteration 3	2	3	13	3 < 3 → false

Table 4.2.11: Iteration-by-iteration values of the counter and accumulator for inputs 4, 7, 2.

In the first iteration x = 4 is read. The accumulator becomes *sum = 0 + 4 = 4*, the counter becomes *count = 0 + 1 = 1*, and the arrow returns to the decision. In the second iteration x = 7 is read; *sum = 4 + 7 = 11*, count = 2. In the third iteration x = 2 is read; *sum = 11 + 2 = 13*, count = 3. At the decision box *3 < 3* is asked; it is false and the loop is exited. The sum is printed: 13. The counter held how many times the loop ran, and the accumulator held the result.

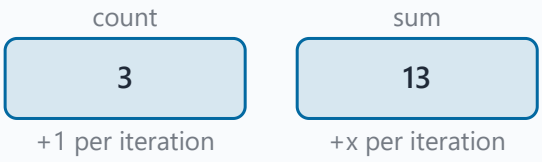


Figure 4.2.19: When the loop ends, the counter is 3 and the accumulator is 13. The counter grew by 1 on each iteration and the accumulator by x.

```java
import java.util.Scanner;

public class SumOfThree {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int count = 0, sum = 0;
        while (count < 3) {
            int x = in.nextInt();          // read x
            sum = sum + x;                 // accumulator
            count = count + 1;             // counter
        }
        System.out.println(sum);
    }
}
```

The program prints 13 for the input 4 7 2 and 12 for 10 −3 5. In the Java code the variable `x` is declared inside the loop. On every iteration a new value is read and the value of x from the previous iteration is no longer needed.

Common mistake

Resetting the accumulator inside the loop. If the `sum = 0` box is mistakenly drawn at the start of the loop body, every iteration begins by erasing the sum of the previous iterations. For inputs 4, 7, 2 the table becomes:

Iteration	x	sum (after reset)	sum (after adding)	count
Iteration 1	4	0	4	1
Iteration 2	7	0	7	2
Iteration 3	2	0	2	3

Trace of the faulty diagram: sum is reset on every iteration.

2 is printed instead of 13; that is, only the last value survives. Boxes that give starting values must run once, before the loop.

4.2.6.4 Sum from 1 to n

A classic example in which a counter and an accumulator are used together is the sum of the numbers from 1 to n. This time the counter does not merely count iterations; it is itself the number to be added.

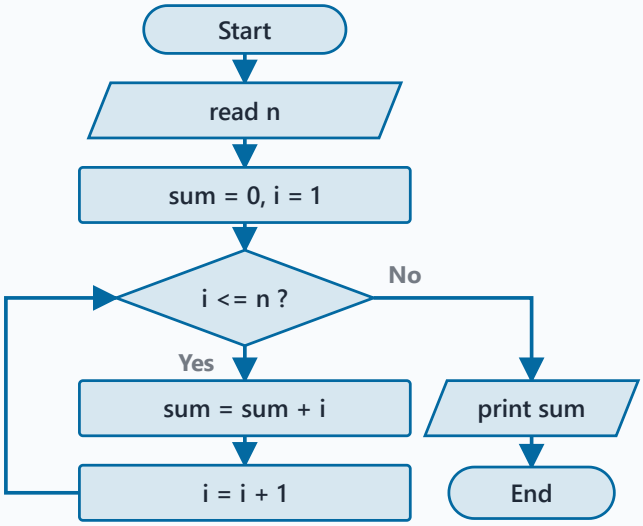


Figure 4.2.20: Sum from 1 to n. n is read; we start with sum = 0, i = 1; while i <= n, sum = sum + i and i = i + 1 are done.

```
Start
  read n
  sum = 0, i = 1
  i <= n ?
    Yes → sum = sum + i
          i = i + 1
          back to the decision
    No → print sum
End
```

In the diagram `i` is the counter and `sum` is the accumulator. Table 4.2.12 shows the execution for n = 4. The expected result is *1 + 2 + 3 + 4 = 10*.

Iteration	i	Decision	sum	i (updated)
Iteration 1	1	1 <= 4 → true	0 + 1 = 1	2
Iteration 2	2	2 <= 4 → true	1 + 2 = 3	3
Iteration 3	3	3 <= 4 → true	3 + 3 = 6	4
Iteration 4	4	4 <= 4 → true	6 + 4 = 10	5
exit	5	5 <= 4 → false	10	—

Table 4.2.12: Trace table of the sum-from-1-to-n diagram for n = 4.

Each row of the table is one iteration. In the first iteration *1 <= 4* is true; sum becomes 0 + 1 = 1 and i rises to 2. In the second iteration sum becomes 1 + 2 = 3, in the third 3 + 3 = 6, in the fourth 6 + 4 = 10. When i = 5, *5 <= 4* is false and the loop is exited. The body ran n = 4 times and the decision n + 1 = 5 times. 10 is printed on the screen.

The result can also be checked with a formula. Writing the numbers from 1 to n once in increasing and once in decreasing order, one under the other, and adding column by column gives the formula step by step. For n = 4:

increasing	1	2	3	4
decreasing	4	3	2	1
column sum	5	5	5	5

Writing the sum from 1 to 4 twice

The sum of each column is $n + 1 = 5$, and there are n = 4 columns. The sum of the two rows is $4 \times 5 = 20$. This is twice the sum we are looking for; the sum is $20 \div 2 = 10$. The general formula is $n(n + 1) / 2$. The formula confirms, by an independent route, that the diagram works correctly.

What happens if n = 0 is entered? With i = 1, the question $1 <= 0$ is false at the first try. The body never runs and 0 is printed. This is also the right answer: no numbers have been added. The formula gives $0 \times 1 / 2 = 0$ as well. The pre-test loop's property that “the body may never run” is useful here.

4.2.6.5 Counter-controlled and sentinel loops

In the loops so far, the number of iterations was known in advance: 5 times, 3 times, n times. In some jobs this number is not known. The cashier does not know in advance how many items will be scanned; when the items run out, they press the “done” key. A teacher entering grades, instead of counting the grades, types a special value at the end. Such a special value is called a sentinel value.

Definition 4.2.15

A **counter-controlled loop** is a loop whose number of repetitions is known in advance and controlled with a counter. A **sentinel value** is a special value that signals the end of the data and is chosen so that it cannot occur in the real data. A **sentinel-controlled loop** is a loop that runs until the sentinel value is read.

	Controlled by a counter	**Controlled by a sentinel**
Number of repetitions	Known in advance (n times)	Unknown; stops when a special value arrives
Decision	`i <= n ?`	`x == -1 ?` (has the sentinel arrived?)
Example	Read the grades of the 30 students in a class	Items scanned at a checkout; until the “done” key
In Java, usually	`for`	`while`

Table 4.2.13: Comparison of counter-controlled and sentinel-controlled loops.

The choice of sentinel depends on the data. If grades are between 0 and 100, −1 is a suitable sentinel, because no student scores −1. 0 is not suitable: a student's grade really can be 0, and that grade would be mistakenly interpreted as “data finished.”

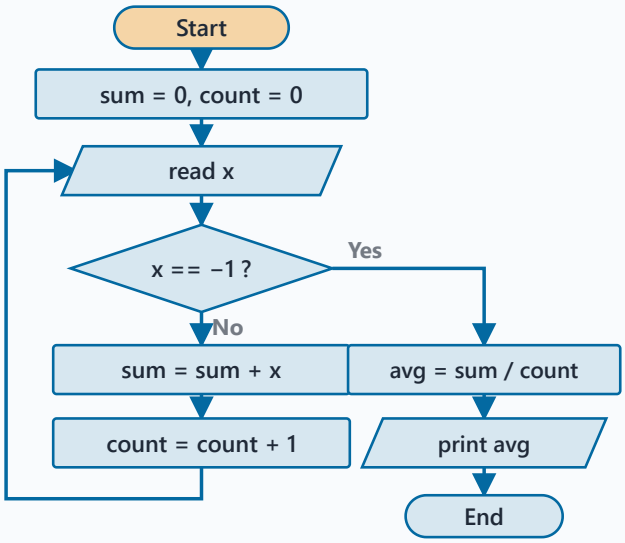


Figure 4.2.21: Sentinel-controlled loop: read until x == −1, add, count; then print the average. The returning arrow goes back to the read box.

```
Start
  sum = 0, count = 0
  read x
  x == -1 ?  Yes → go to line 7
  sum = sum + x
  count = count + 1;  back to line 3
  avg = sum / count
  print avg
End
```

Figure 4.2.21 shows two important details of the sentinel-controlled loop. First, the decision box comes right after the read box, *before* adding and counting. This way, when the sentinel is read, we leave the loop without it being added or counted. Second, the returning arrow goes back not to the decision box but to the front of the read box. A new number must be read on every iteration; if the arrow went back to the decision, the same x would be processed over and over.

x read	x == −1 ?	sum	count
80	No	80	1
64	No	144	2
90	No	234	3

x read	x == −1 ?	sum	count
−1	Yes (sentinel)	234	3

Table 4.2.14: Trace table of the sentinel-controlled loop for inputs 80, 64, 90, −1.

Table 4.2.14 gives the execution for inputs 80, 64, 90, −1. Each grade read is first compared with the sentinel. 80, 64, and 90 are real grades; they are added to the sum and counted. When −1 is read, the decision is Yes and the loop ends. The average is *234 / 3 = 78*. The body ran 3 times and the decision 4 times. How many grades would be entered was not known in advance; the data itself stopped the loop.

If the first input is the sentinel, the body never runs and the counter stays 0. In that case the average computation becomes a division by zero. That is why `count > 0 ?` is asked before the division. The Java counterpart includes this check as well:

```
import java.util.Scanner;

public class GradeAverage {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int sum = 0, count = 0;
        int x = in.nextInt();             // the first grade is read before the loop
        while (x != -1) {                 // has the sentinel arrived?
            sum = sum + x;
            count = count + 1;
            x = in.nextInt();             // next grade: the returning arrow goes to the
read box
        }
        if (count > 0) {
            double avg = (double) sum / count;
            System.out.println(avg);
        } else {
            System.out.println("No grades entered");
        }
    }
}
```

In the Java code the first grade is read once before the loop, and the later grades are read at the end of the body. This is how the diagram's detail “the returning arrow goes to the read box” is written with `while`. Because the average can have a fractional part, it is computed as a `double`; the expression `(double) sum` converts the sum to a floating-point number before the division. The program prints `78.0` for 80 64 90 −1, `62.5` for 55 70 −1, and “No grades entered” for −1 alone.

Common mistake

Testing the sentinel after processing. If the decision box is drawn below the add and count boxes, −1 is also processed as data:

x read	sum	count	x == −1 ?
80	80	1	No
64	144	2	No
90	234	3	No
−1	233	4	Yes

Faulty order: first add and count, then ask about the sentinel.

The sum becomes 233 instead of 234, the counter 4 instead of 3, and the average comes out as *233 / 4 = 58.25* instead of 78. The decision about the sentinel must come before every operation that uses the value.

4.2.6.6 Nested loops

The single-entry, single-exit rule of structured programming allows one structure to be placed inside another. A decision can contain another decision on one of its branches. A loop body can contain a second loop.

Definition 4.2.16

A **nested loop** is a loop that has another loop in its body. The outer one is called the **outer loop** and the inner one the **inner loop**. On every iteration of the outer loop, the inner loop runs from start to finish.

The multiplication table is the most familiar example of a nested loop. The outer loop moves through the rows (i), the inner loop through the columns (j); i × j is written in each cell. Figure 4.2.22 shows the 3 × 3 table. The outer loop runs 3 iterations, and the inner loop runs 3 iterations on each outer iteration. The inner body runs a total of *3 × 3 = 9* times.

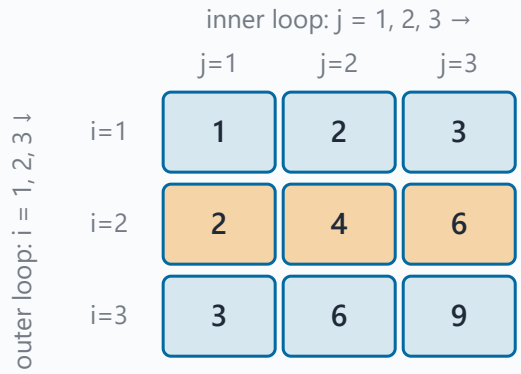


Figure 4.2.22: The nested-loop grid. The outer loop moves through the rows (i = 1, 2, 3), the inner loop through the columns (j = 1, 2, 3); the cells hold i × j. The highlighted row is the three iterations of j while i = 2.

To keep the diagram small, a 2 × 2 table is drawn. Figure 4.2.23 shows how the two loops fit together.

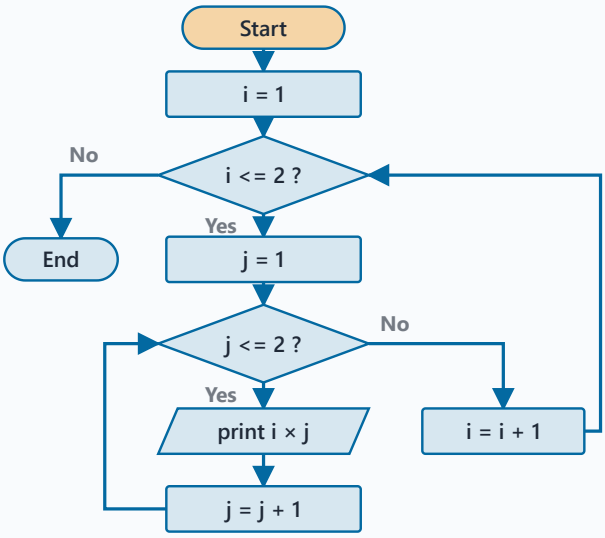


Figure 4.2.23: Nested loops: i from 1 to 2, j from 1 to 2; the inner body prints i × j. When the inner loop ends, i increases and we return to the outer decision.

```
Start
  i = 1
  i <= 2 ?  (No → End)
    j = 1
    j <= 2 ?  (No → i = i + 1)
      print i × j
      j = j + 1, back to inner decision
    end line (println), i = i + 1, back to outer decision
End
```

The body of the outer loop consists of three parts: `j = 1`, the whole inner loop, and `i = i + 1`. It matters that the `j = 1` box is inside the outer body. On every outer iteration j becomes 1 again; this way the inner loop starts each row from the beginning.

i	j	Inner decision	Action
1	1	1 <= 2 → true	1 × 1 = 1
1	2	2 <= 2 → true	1 × 2 = 2
1	3	3 <= 2 → false	end of line
2	1	1 <= 2 → true	2 × 1 = 2
2	2	2 <= 2 → true	2 × 2 = 4
2	3	3 <= 2 → false	end of line

Table 4.2.15: Trace table of the 2 × 2 nested loop. Each row is one inner-decision question.

Table 4.2.15 gives the execution. With i = 1 the outer decision *1 <= 2* is true. j = 1; the inner decision is true and 1 × 1 = 1 is printed. j = 2; 1 × 2 = 2 is printed. When j = 3 the inner decision is false and the inner loop ends. The end of line is printed, i = 2, and we return to the outer decision. In the second outer iteration j starts again at 1 and the same path is followed: 2 and 4 are printed. When i = 3 the outer decision *3 <= 2* is false and we go to End. The output is a small multiplication table:

```
1 2
2 4
```

The inner body ran *2 × 2 = 4* times. In general, if the outer loop runs n iterations and the inner loop m, the inner body runs n × m times. As the numbers grow, this product grows quickly: in a 1000 × 1000 table the inner body runs a million times.

```
public class MultiplicationTable {
    public static void main(String[] args) {
        int i = 1;
        while (i <= 2) {                  // outer decision
            int j = 1;                    // inner counter is 1 again on every outer
iteration
            while (j <= 2) {              // inner decision
                System.out.print(i * j + " ");
                j = j + 1;
            }
            System.out.println();         // end of line
            i = i + 1;
        }
    }
}
```

The program prints two lines: `1 2` and `2 4`. In the Java code the line `int j = 1;` is inside the outer loop's braces. If this line is moved before the outer loop, j stays at 3 in the second outer iteration, the inner decision comes out false at the first try, and the second line is printed empty.

A large number of nested structures can be a sign that a part of the algorithm should be drawn as a separate subprocess. Three or four levels of nested decisions and loops make both the diagram and the code unreadable.

4.2.6.7 The number-guessing game

The computer holds a secret number between 1 and 100. The player guesses; the computer gives a hint, “Higher” or “Lower.” The game ends when the guess is right, and the number of tries is printed. The player's guesses determine how many iterations the loop runs; in this respect the game is a kind of sentinel-controlled loop.

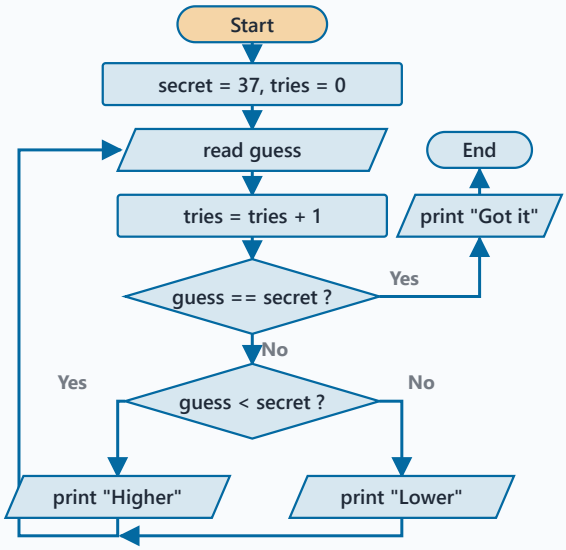


Figure 4.2.24: The number-guessing game: the guess is read again, with a hint, until it equals the secret number. On a correct guess “Got it” is printed and the game ends.

```
Start
  secret = 37, tries = 0
  read guess
  tries = tries + 1
  guess == secret ?
    Yes → print "Got it (try " tries ")", End
  guess < secret ?
    Yes → print "Higher"
    No → print "Lower"
  back to line 3 (read)
End
```

There are two decisions in the diagram. The first is the question `guess == secret ?` and it ends the loop. The second is the question `guess < secret ?` and it decides which hint is given. The second decision is on the No branch of the first; if the guess is right, the hint question is never asked. After the two hint boxes, the arrow returns to the read box. The secret number is normally chosen at random; here 37 is used so that the trace can be repeated.

guess	tries	guess == 37 ?	guess < 37 ?	Output
50	1	No	No	Lower
25	2	No	Yes	Higher
37	3	Yes	not asked	Got it (try 3)

Table 4.2.16: Trace table for secret number 37 and guesses 50, 25, 37.

The player first enters 50; tries = 1. *50 == 37* is false, so we move to the hint question. *50 < 37* is also false; since the guess is too high, “Lower” is printed. The second guess is 25; *25 < 37* is true and “Higher” is printed. The third guess is 37; the equality is true and “Got it (try 3)” is printed.

```
import java.util.Scanner;

public class NumberGuess {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int secret = 37;                  // in the game: new
java.util.Random().nextInt(100) + 1
        int tries = 0;
        int guess = in.nextInt();
        tries = tries + 1;
        while (guess != secret) {         // guess == secret ? the No branch stays in the
loop
            if (guess < secret) {
                System.out.println("Higher");
            } else {
                System.out.println("Lower");
            }
            guess = in.nextInt();
            tries = tries + 1;
        }
        System.out.println("Got it (try " + tries + ")");
    }
}
```

The Java program writes the diagram with a slightly different layout. The first guess is read before the loop, and the loop condition becomes `guess != secret`. The next guess is read at the end of the body. This is the same layout as in the sentinel-controlled loop. With the input 50 25 37 the program prints “Lower,” “Higher,” and “Got it (try 3)”; with 20 40 37 it prints “Higher,” “Lower,” and “Got it (try 3).” If 37 is entered on the first guess, the loop body never runs and “Got it (try 1)” is printed.

The game has a good strategy: guess the middle of the remaining range each time. In the range 1–100 the first guess is 50; if the hint is “Lower,” the range drops to 1–49, and if “Higher,” to 51–100. Each guess splits the range roughly in half. In the worst case the range shrinks as 100, 50, 25, 12, 6, 3, 1; so 7 guesses are always enough. The reason is that $2^7 = 128$ is greater than 100.

4.2.6.8 The post-test loop

In some jobs the loop body must run at least once. The user is asked for a number between 1 and 5; if the number entered is invalid, the question is asked again. Whether the number is valid cannot be asked before it is read. In this case it is more natural to put the condition at the end of the body.

Definition 4.2.17

A **post-test loop** is a loop in which the condition is asked after the body. The body runs first and then the condition is examined; so the body runs *at least once*. In Java it is written in the form `do { … } while (condition);`.

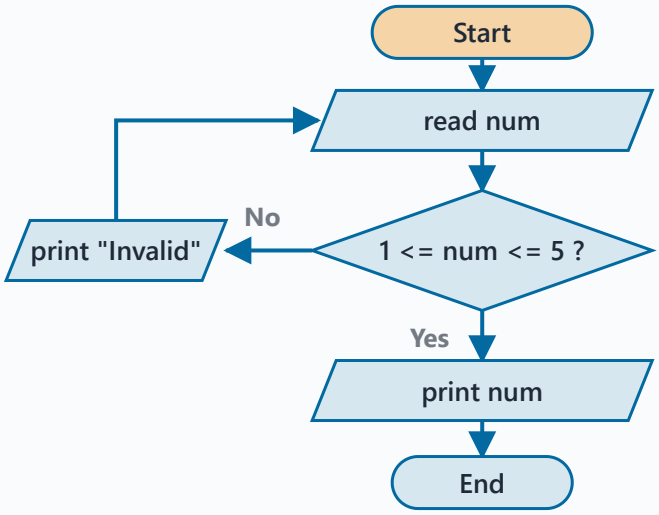


Figure 4.2.25: Input validation with a post-test loop: num is read; if it is not 1 ≤ num ≤ 5, “Invalid” is printed and it is read again.

```
Start
  read num
  1 <= num <= 5 ?
    No → print "Invalid", back to read
    Yes → print num
End
```

In the diagram of Figure 4.2.25, after Start execution goes directly into the read box; there is no decision in between. The decision comes after the reading. On an invalid number the No branch prints “Invalid” and the arrow returns to the read box. This kind of use is called **input validation**.

Iteration	Number read	1 ≤ num ≤ 5 ?	Output
1	7	false → No	Invalid
2	0	false → No	Invalid
3	3	true → Yes	Valid: 3

Table 4.2.17: Trace table of the post-test loop for inputs 7, 0, 3.

In the first iteration 7 is read before the condition has been asked even once. Then comes the decision: is 1 ≤ 7 ≤ 5? Since 7 is greater than 5, the answer is false; “Invalid” is printed. In the second iteration 0 is read; since 0 is less than 1, it is invalid again. In the third iteration 3 is read; 1 ≤ 3 ≤ 5 is true and “Valid: 3” is printed. The body ran 3 times and the condition was also asked 3 times. Unlike a pre-test loop, here the number of decisions equals the number of body runs.

The condition `1 <= num <= 5` in the diagram is mathematical notation. Java does not accept this form; the two comparisons are joined with a compound condition. The invalidity condition becomes “num is less than 1 OR greater than 5”:

```
import java.util.Scanner;

public class InputValidation {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int num;
        do {
            num = in.nextInt();                    // body: read first
            if (num < 1 || num > 5) {
                System.out.println("Invalid");
            }
        } while (num < 1 || num > 5);           // the condition is asked at the end
        System.out.println("Valid: " + num);
    }
}
```

With the input 7 0 3 the program prints “Invalid” twice and then “Valid: 3.” If the first input is 4, the body runs once and “Valid: 4” is printed directly. In the code the invalidity condition is written twice: once inside the `if` for the warning, once in the loop condition. The single decision box in the diagram becomes two separate questions in the code.

Common mistake

Thinking the difference between `while` and `do-while` is only a difference of notation. The difference shows up when the condition is false at the first try. If the sum-from-1-to-n diagram were drawn as a post-test loop, then for n = 0 the body would run once and the sum would be 1; yet the right answer is 0. Whenever the body may need to never run, a pre-test loop is chosen.

4.2.6.9 The flowchart of a for loop

Counter-controlled loops always contain the same four parts: the counter's starting value, the condition, the body, and the update of the counter. Java offers a shorter notation for this pattern: the `for` loop.

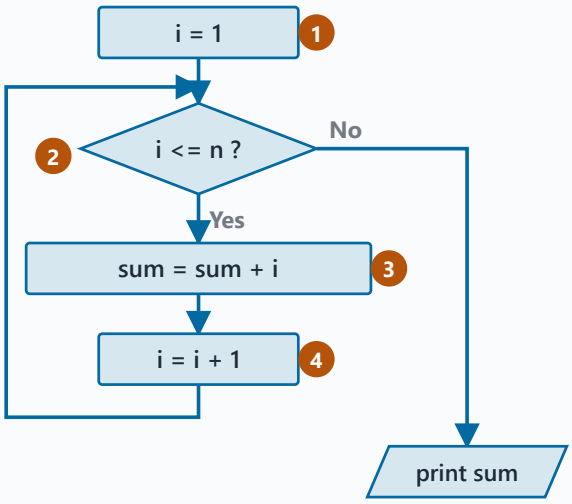


Figure 4.2.26: The four parts of a for loop: (1) initialization i = 1, (2) condition i <= n, (3) body sum = sum + i, (4) update i = i + 1.

```
for (i = 1; i <= n; i++) {
    sum = sum + i;
}
```

1. **Initialization** (`i = 1`): runs only once, when entering the loop.
2. **Condition** (`i <= n`): asked before every iteration; a for loop is a pre-test loop.
3. **Body**: runs if the condition is true.
4. **Update** (`i++`): runs after the body; then the arrow returns to the condition. `i++` is a short way of writing `i = i + 1` .

Figure 4.2.26 is no different from a while diagram. A for loop merely gathers parts 1, 2, and 4 on one line. Some sources show these three parts in the preparation symbol among the additional symbols (a hexagon, “i = 1 → n”).

```
public class ForSum {
    public static void main(String[] args) {
        int n = 4, sum = 0;
        for (int i = 1; i <= n; i++) {    // initialization; condition; update
            sum = sum + i;                // body
        }
        System.out.println(sum);
    }
}
```

The program prints 10 for n = 4; it gives the same result as the sum-from-1-to-n program written with while. Which loop to choose is a matter of readability. When the number of repetitions is known in advance, `for` reads more clearly; when the data stops the loop, `while` does.

4.2.6.10 Early exit, flag, and jump

Suppose we are looking for the first negative number in an array: a = [4, 7, −2, 5, −8]. Once the number sought is found, the remaining iterations are unnecessary. A second arrow going out of the loop from inside the body ends the loop at once.

Definition 4.2.18

An **early exit** is leaving a loop from inside the body before the loop's condition has become false; in Java it is done with the `break` statement. A **flag** is a true/false variable that records whether an event has happened. **Continue**, on the other hand, does not leave the loop; it skips the rest of that iteration and returns to the update and the condition.

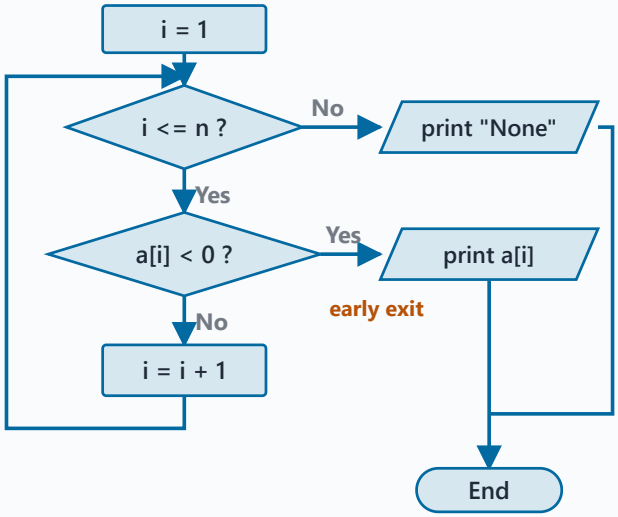


Figure 4.2.27: A loop that looks for the first negative number. The loop has two exits: one from the condition (i <= n false: print “None”) and one from the body (a[i] < 0: print a[i], early exit).

In the diagram the elements of the array are written `a[i]`: a[1] is the first element of the array and a[2] the second. An array is an ordered list of values of the same type; here there are five numbers, so n = 5.

i	a[i]	i <= n ?	a[i] < 0 ?	Done
1	4	1 <= 5 → Yes	No	i = 2
2	7	2 <= 5 → Yes	No	i = 3
3	−2	3 <= 5 → Yes	Yes	print −2, leave the loop

Table 4.2.18: Search for the first negative number for a = [4, 7, −2, 5, −8].

At i = 1, 4 and at i = 2, 7 are not negative. At i = 3, −2 is found and printed; the loop ends by early exit. 5 and −8 are never looked at. If there were no negative number in the array, the loop would exit through the condition at i = 6 and “None” would be printed.

A loop with an early exit has two exits, and this does not obey the single-exit rule of structured programming. If a single exit is wanted, a flag is used. We start with `found = false`, set `found = true` when the target is found, and the loop condition becomes `i <= n AND not found ?`. When the loop ends, the flag is checked to print the result. In Java both ways are used:

```
public class FirstNegative {
    public static void main(String[] args) {
        int[] a = {4, 7, -2, 5, -8};
        // way 1: early exit with break (a loop with two exits)
        int i = 0;
        while (i < a.length) {
            if (a[i] < 0) {
                break;                    // target found: leave the loop
            }
            i = i + 1;
        }
        if (i < a.length) {
            System.out.println(a[i]);
        } else {
            System.out.println("None");
        }
        // way 2: a single-exit loop with a flag
        boolean found = false;
        int k = 0;
        while (k < a.length && !found) {
            if (a[k] < 0) {
                found = true;
            } else {
                k = k + 1;
            }
        }
        System.out.println(found ? a[k] + "" : "None");
    }
}
```

Both ways print −2. In Java, array elements are numbered from 0: `a[0]` is 4 and `a[2]` is −2. That is why in the code the counter starts at 0 and the condition is `i < a.length`; `a.length` is the number of elements in the array (5). In the flag version, the expression `!found` means “not found.” The condition `k < a.length && !found` is a compound condition and works with short-circuit evaluation.

Continue, on the other hand, does not end the loop; it only skips the rest of that iteration. For example, if the sum of the non-negative numbers in the array is wanted, the adding step is skipped when a negative number is reached:

```java
public class ContinueExample {
    public static void main(String[] args) {
        int[] a = {4, 7, -2, 5, -8};
        int sum = 0;
        for (int i = 0; i < a.length; i++) {
            if (a[i] < 0) {
                continue;                // skip the negative, go to the next iteration
            }
            sum = sum + a[i];
        }
        System.out.println(sum);
    }
}
```

The program prints *4 + 7 + 5 = 16*. At −2 and −8, `continue` runs; the adding line is skipped, `i++` runs, and we return to the condition. In a diagram, continue is shown with an arrow from the middle of the body to the update box.

4.2.7 From flowchart to Java code

The value of a flowchart comes from its being turned, in the end, into a running program. In the earlier sections the Java counterpart of each structure was given separately. This section gathers the translation into a list of rules and applies it line by line to one complete example.

4.2.7.1 Code counterparts of the symbols

Flowchart element	Java counterpart
Start / End	The opening and closing braces of the `main` method
Input (read)	`int a = in.nextInt();` or a variable declaration with a fixed value
Process (assignment)	An assignment statement: `sum = sum + i;`
Output (print)	`System.out.println(…);`
Decision, two branches merging	`if (condition) { … } else { … }`
Decision chain	`if … else if … else`
Decision + arrow returning from the body (condition first)	`while (condition) { … }` or `for`
Body + arrow returning to the decision (condition last)	`do { … } while (condition);`
Arrow from the body to outside the loop	`break;`
Predefined process	Method call: `average(a, b, c)`

Table 4.2.19: Java counterparts of flowchart elements.

The most important of the rules is the loop translation. If there is an arrow that starts from the Yes branch of a decision box and returns to the same decision, that decision is a `while` condition and the boxes up to the arrow are the loop body. To find which boxes the body ends with, look at where the returning arrow leaves from.

4.2.7.2 Sum from 1 to n in Java

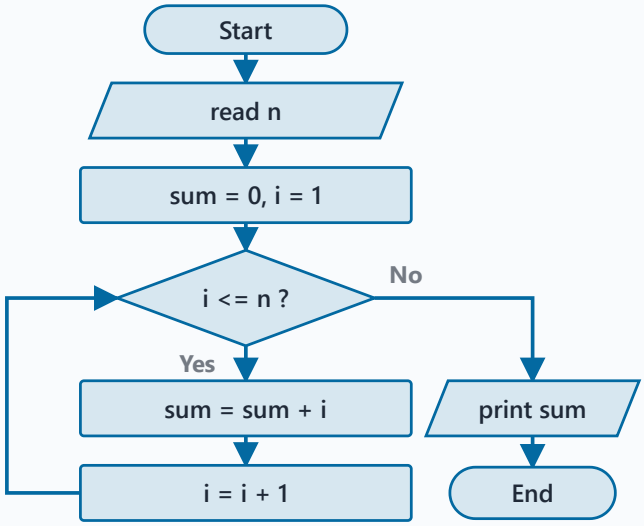


Figure 4.2.28: The diagram to be translated into Java: sum from 1 to n.

```
public static void main(String[] args) {
    int n = 3;
    int sum = 0, i = 1;
    while (i <= n) {
        sum = sum + i;
        i = i + 1;
    }
    System.out.println(sum);
}
```

The translation is done box by box from top to bottom:

1. The Start symbol is the beginning of the method: `main` opens.
2. “read n” is an input symbol. To keep the example short, the input is taken as a constant: `int n = 3;`. If reading from the keyboard is wanted, this line becomes `int n = in.nextInt();`.
3. The process symbol “sum = 0, i = 1” gives the starting values of two variables: `int sum = 0, i = 1;`.
4. From the decision `i <= n ?`, the Yes branch goes to two boxes and an arrow returns to the decision. This is a loop: `while (i <= n)`.

5. The two rectangles on the Yes branch are the loop body. Both are written inside the braces, in the order shown in the diagram.
6. The returning arrow is the closing brace of the body block. Execution returns from there to the condition.
7. The output symbol “print sum” on the No branch is outside the loop: `System.out.println(sum);`.
8. The End symbol is the end of the method.

When the program is run for n = 3, execution follows the same path as the diagram, step by step. The first try of the condition, *1 <= 3*, is true; sum = 0 + 1 = 1 and i = 2. Then sum becomes 3 and 6. When i = 4, *4 <= 3* is false; the No path is followed and execution leaves the loop block. 6 is printed on the screen. The result is also verified by the formula: *3 × 4 / 2 = 6*.

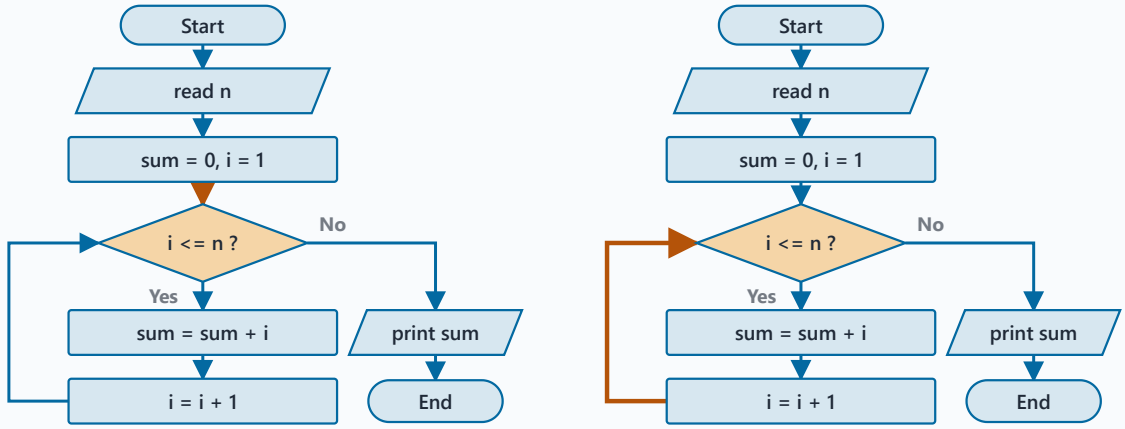


Figure 4.2.29: The code counterpart of the decision symbol. (a) First try: while (i <= n), 1 <= 3 is true; the body is entered. (b) Last try: 4 <= 3 is false; the No path leaves the loop block.

At the end of the translation, every symbol has a code counterpart and every line of code has a symbol counterpart. This one-to-one correspondence is the easiest way to check that the diagram has been translated correctly. If there is a symbol with no counterpart in the program, or a line with no counterpart in the diagram, something is missing or extra in the translation.

Common mistake

Writing one of the body boxes outside the loop block. If `while (i <= n) { sum = sum + i; } i = i + 1;` is written, the counter's increment is left outside the loop. i never changes, the condition always comes out true, and the program enters an infinite loop. In the diagram, every box on the Yes branch, up to the last box before the returning arrow leaves, is inside the braces.

4.2.7.3 Modular diagrams with a subprocess

As diagrams grow, they no longer fit on a single page. It also happens that a job is repeated in several places. In that case part of the work is drawn as a separate diagram and called from the main diagram.

Definition 4.2.19

A **subprocess** is a separately drawn diagram referred to by a name. In the main diagram it is called with the **predefined process** symbol. When the subprocess ends, the flow continues in the main diagram from the point where it was called. In Java every subprocess becomes a **method**.

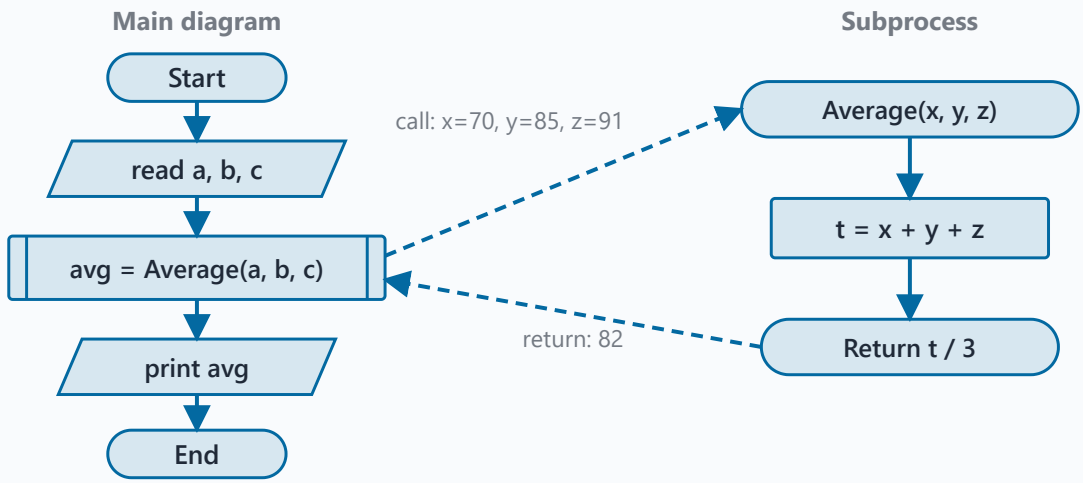


Figure 4.2.30: The main diagram calls the subprocess, and the subprocess returns a value. For a = 70, b = 85, c = 91, Average(70, 85, 91) is called and 82 is returned.

The sub-diagram in Figure 4.2.30 begins not with Start but with its name and parameters, "Average(x, y, z)." It ends not with End but with "Return." Parameters are the values the subprocess receives from the place it is called. In the box `avg = Average(a, b, c)` the main diagram sends the values of a, b, and c to x, y, and z. For a = 70, b = 85, c = 91, the subprocess computes *t = 70 + 85 + 91 = 246* and returns *246 / 3 = 82*. The main diagram writes this value into the variable avg and continues from the next box.

A subprocess has two benefits. First, the same subprocess can be called from many places but is drawn only once. If an error is found in the average computation, it is fixed in a single place. Second, the main diagram becomes shorter and says "what is done"; "how it is done" stays in the sub-diagram.

```java
import java.util.Scanner;

public class Subprocess {
    // subprocess: Average(x, y, z)
    static int average(int x, int y, int z) {
        int t = x + y + z;
        return t / 3;                    // Return t / 3
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int a = in.nextInt(), b = in.nextInt(), c = in.nextInt();
        int avg = average(a, b, c);      // predefined process symbol
        System.out.println(avg);
    }
}
```

The line `static int average(int x, int y, int z)` is the heading of the sub-diagram: the method's name, its three parameters, and the type of the value it returns (`int`). The line `return t / 3;` is the counterpart of the Return symbol. With the input 70 85 91 the program prints 82.

Common mistake

Forgetting integer division. In Java the quotient of two integers is also an integer; the fractional part is discarded. For 70, 85, 91 the sum 246 divides exactly by 3, so there is no visible problem. For 50, 60, 71 the sum is 181; the true average is about 60.33, but the program prints 60. If a fractional result is needed, the method must return `double` and the division must be written `t / 3.0` .

4.2.8 Verification

That a diagram is correct is tested by executing it by hand on a few inputs. Which inputs to choose is not arbitrary. A small number of well-chosen inputs catch more errors than a large number of badly chosen ones.

4.2.8.1 Testing a flowchart

Definition 4.2.20

A **test case** is a pair of an input and the output expected for it. The diagram is executed with the input and the output found is compared with the expected one. A **boundary value** is an input at the point where the answer to a condition changes; for example, 60 for `grade >= 60` .

Test inputs are chosen by five rules:

1. A normal input is chosen and the arrows are followed step by step.
2. Inputs are chosen that run every exit of every decision. Every box in the diagram must run at least once.
3. Boundary values are tried separately.
4. For each loop, cases are tried in which the body runs zero, one, and many times.
5. Paths that do not return to the beginning or do not reach End are looked for.

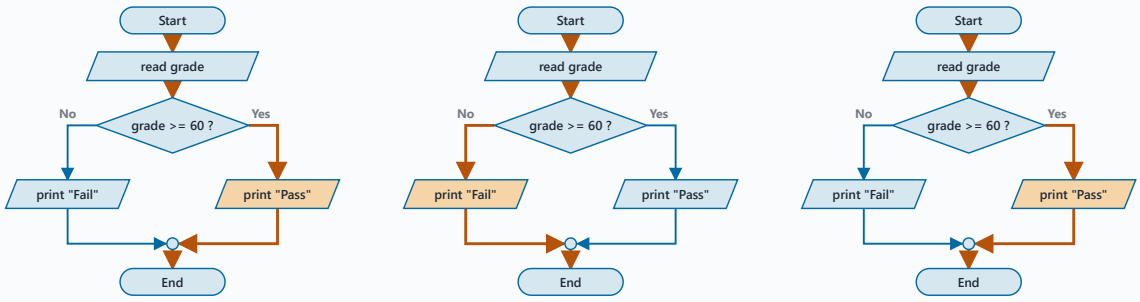


Figure 4.2.31: Testing the Pass/Fail diagram. (a) Normal input grade = 72: the Yes path. (b) grade = 45: the No path; now every box has been tried at least once. (c) Boundary value grade = 60: 60 >= 60 is true, "Pass."

Figure 4.2.31 shows the rules applied to the Pass/Fail diagram. 72 is a normal input and runs the Yes path. 45 runs the other branch; with the two inputs every box has been tried. 60 is the boundary value and would be the only input to expose the error had `>` been written instead of `>=` in the decision box.

The reason for rule 4 on loops is this: loop errors often hide at the ends. If the accumulator's starting value is wrong, this shows most clearly in the case where the body never runs. A `<` / `<=` mix-up in the condition usually causes one iteration too few or too many and is easy to notice in a one-iteration case. Table 4.2.20 gives these three cases for the sum-from-1-to-n diagram.

n	Case	How many times the body runs	Expected (n(n+1)/2)	Output of the diagram
0	zero repetitions	0	0	0
1	one repetition	1	1	1
5	many repetitions	5	15	15

Table 4.2.20: Loop tests for the sum-from-1-to-n diagram. The expected value comes from the formula, the output from executing the diagram.

In all three rows the output equals the expected value. Had the condition been mistakenly written `i < n`, then for n = 1 the body would never run and 0 would be printed; for n = 5, 1 + 2 + 3 + 4 = 10 would be printed. The error shows immediately on two of the three inputs. A negative n is an invalid input. The diagram prints 0 in this case too; whether this is the desired behavior depends on the problem definition.

4.2.8.2 Hand tracing with a trace table

The trace table has been used often in this chapter. The pattern can be summed up briefly: each variable is a column and each iteration is a row; the result of the decision is also written in a separate column. A table is indispensable especially in algorithms where variables change in a mutually dependent way. Euclid's algorithm is a nice example.

Euclid's algorithm finds the **greatest common divisor** (GCD) of two numbers. The idea is this: the common divisors of a and b are the same as the common divisors of b and the remainder of a divided by b. So the pair (a, b) is replaced by the pair (b, a % b). The remainder shrinks on every iteration and finally becomes 0. At that moment a is the GCD we are looking for.

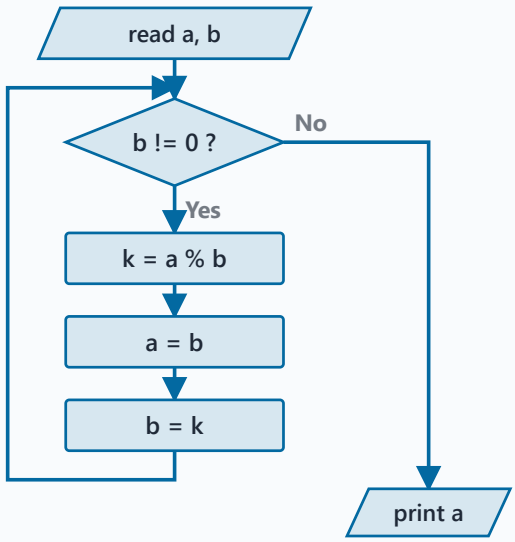


Figure 4.2.32: Euclid's algorithm: until b is zero, k = a % b, a = b, b = k are done; at the end a is printed.

State	a	b	k	b != 0 ?
start	48	18	—	18 != 0 → true
Iteration 1	18	12	12	12 != 0 → true
Iteration 2	12	6	6	6 != 0 → true
Iteration 3	6	0	0	0 != 0 → false

Table 4.2.21: Trace table of Euclid's algorithm for (48, 18). The last column is the next decision asked with the values in that row.

Initially a = 48, b = 18 and *18 != 0* is true. In the first iteration *k = 48 % 18 = 12* (48 = 2 × 18 + 12); a = 18, b = 12. In the second iteration *k = 18 % 12 = 6*; a = 12, b = 6. In the third iteration *k = 12 % 6 = 0*; a = 6, b = 0. The decision *0 != 0* is false; the loop is exited and 6 is printed. GCD(48, 18) = 6. Check: *48 = 8 × 6*, *18 = 3 × 6*, and 8 and 3 have no common divisor greater than 1.

The loop ran 3 iterations and the decision was asked 4 times: as in every pre-test loop, the decision ran one more time than the body. The table also shows the importance of order in the diagram. What would happen if the boxes `a = b` and `b = k` in the body were swapped, that is, if `b = k` were done first and then `a = b`?

State	a	b	k	b != 0 ?
start	48	18	—	18 != 0 → true
Iteration 1	12	12	12	12 != 0 → true
Iteration 2	0	0	0	0 != 0 → false

Table 4.2.22: Trace for (48, 18) when b = k and a = b are swapped in the body.

In the first iteration b becomes 12, and then with a = b, a also becomes 12. a has taken b's new value, not its old one. In the second iteration *12 % 12 = 0* is found and both become 0. The result comes out 0 instead of 6. This is the same error as the swap error in Example 4.2.2: an assignment has erased a value that was needed later.

```
import java.util.Scanner;

public class Euclid {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int a = in.nextInt(), b = in.nextInt();
        while (b != 0) {
            int k = a % b;
            a = b;
            b = k;
        }
        System.out.println(a);
    }
}
```

The program prints 6 for (48, 18), 6 for (270, 192), and 12 for (36, 84). The algorithm works even when a < b. For (18, 48), in the first iteration *18 % 48 = 18* is found and a and b are swapped; the result is again 6.

4.2.8.3 Common mistakes

The mistakes students make most often in flowcharts can be reduced to a few patterns. Table 4.2.23 collects drawing mistakes and Table 4.2.24 logic mistakes.

Mistake	Result	Fix
Unlabeled decision path	It is unclear which condition selects it	Write Yes/No
Counter that is not updated	Infinite loop	Add the counter step

Mistake	Result	Fix
Unconnected symbol	The flow breaks	Connect with an arrow
Many operations in one symbol	Readability suffers	Split into steps

Table 4.2.23: Drawing mistakes.

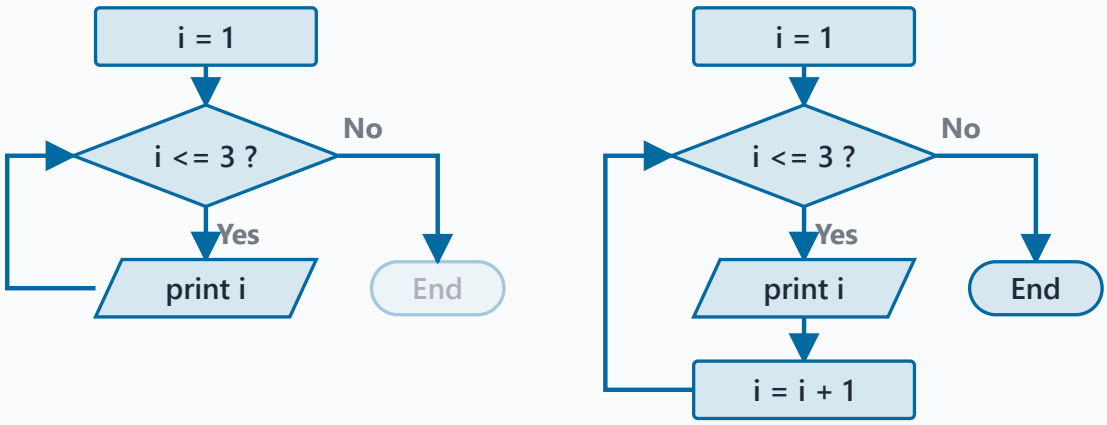


Figure 4.2.33: A counter that is not updated. (a) Faulty: i stays 1, the screen shows 1 1 1 1 ... and End is never reached. (b) Correct: when i = i + 1 is added, 1 2 3 is printed, then we go to End.

Figure 4.2.33 shows the most dangerous drawing mistake. In the diagram on the left, there is no box on the path of the returning arrow that changes i. The condition gives the same answer every time and the loop never ends. When a loop is drawn, the first question to ask is: which box on the repeat path will someday make the condition false?

Mistake	Symptom	Fix
Accumulator is reset inside the loop	For 4, 7, 2, 2 is printed instead of 13: only the last value survives	Move the `sum = 0` box to before the loop
Returning arrow skips the read box	The same x is processed over and over; the sentinel never arrives	The arrow returns to the front of the read box
Assignment in a decision box: `x = 0 ?`	Unclear whether it is a question or an assignment	Write a comparison: `x == 0 ?`
Sentinel is tested after processing	−1 is added to the sum and the counter is one too high	Ask the decision before adding and counting
Wide condition first in a decision chain	The lower branches never run	Order from narrow to wide
Only one arrow leaves a decision	The flow breaks in the other case	Draw both exits, write Yes/No
No zero check before dividing	Division by 0 if no data is entered	Add a `count > 0 ?` decision

Table 4.2.24: Logic mistakes, their symptoms, and their fixes.

Most of the mistakes in the table were shown with trace tables in the earlier sections of this chapter. What they have in common is that all are noticed immediately when the diagram is executed by hand on at least one suitable input. The accumulator mistake shows up with the input 4, 7, 2, the sentinel mistake with 80, 64, 90, −1. The chain-order mistake is exposed by −5, and the missing zero check by an input consisting of −1 alone. Executing a diagram on a few well-chosen inputs after drawing it is the most effective safeguard against all of these mistakes.

4.2.9 Summary

- A **flowchart** shows the steps of an algorithm with standard symbols and the order of execution with arrows. It is independent of the programming language; it lets us discuss the logic before the code and see errors early.
- There are five basic symbols: **start/end** (oval), **process** (rectangle: computation, assignment), **decision** (diamond: a Yes/No question), **input/output** (parallelogram: read, print), and the **arrow**. Paths merge at a **connector**. Additional symbols (off-page connector, predefined process, preparation, document, data store, annotation) are used in large diagrams.
- A decision symbol gets a question, a process symbol a command. Assignment (`=`) and the equality question (`==`) are different.
- A good diagram flows from top to bottom, labels the decision exits, and does one job per box. The same algorithm can also be described in pseudocode; the two notations translate into each other line by line.
- According to the **Böhm–Jacopini theorem**, every algorithm can be written with only the structures of **sequence**, **selection**, and **repetition**. Each structure has a single entry and a single exit; that is why structures can be nested.
- In **sequential flow**, every box runs exactly once and in order. The order of the boxes changes the result; a swap needs a temporary variable.
- In **conditional flow**, only one of the branches is followed on each run. With many options, decisions are arranged as a **decision chain**, from the narrow condition to the wide one; n options need n − 1 decisions. In a **compound condition**, AND and OR, written `&&` and `||` in Java, are evaluated with **short-circuit evaluation**.
- In **repetitive flow**, the returning arrow forms the loop. In a pre-test loop the decision runs one more time than the body; if there is no operation on the repeat path that changes the condition, an **infinite loop** results.
- The **counter** holds the number of iterations and the **accumulator** the partial result; the accumulator starts at 0 for a sum and 1 for a product, before the loop. A **sentinel value** is a value that cannot occur in the real data; its decision is asked before the boxes that process the data.
- In a **nested loop**, the inner loop runs from the start on every iteration of the outer loop; the inner body runs n × m times. In a **post-test loop** (`do-while`), the body runs at least once. `for` is a pre-test loop that gathers initialization, condition, and update on one line.

- An **early exit** (`break`) adds a second exit to a loop; if a single exit is wanted, a **flag** is used. **Continue** (`continue`) only skips the rest of that iteration.
- A diagram is translated into Java symbol by symbol: read → `nextInt()` , process → assignment, decision → `if` , decision + returning arrow → `while` , print → `println` , **subprocess** → method.
- A diagram is tested with inputs that run every exit of every decision, with **boundary values**, and with cases where the loop runs zero, one, and many times. In a **trace table**, each variable is a column and each iteration a row.

4.2.10 Key terms

The key terms of this chapter are listed below.

flowchart terminal symbol process symbol decision symbol input/output symbol flow line connector off-page connector predefined process preparation assignment variable condition pseudocode structured programming spaghetti code sequential flow temporary variable conditional flow branch multi-way decision compound condition short-circuit evaluation remainder repetitive flow loop loop body iteration pre-test loop post-test loop infinite loop counter accumulator counter-controlled loop sentinel value sentinel-controlled loop nested loop input validation early exit flag continue subprocess method test case boundary value trace table greatest common divisor (GCD)

4.2.11 Test Yourself

1. Which symbol is each of the following steps drawn with? (a) `avg = sum / 3` (b) Is the number negative? (c) Print the result on the screen. (d) Use the “Compute grade average” process drawn on a separate page.

   Answer

   (a) is an assignment: process symbol, rectangle. (b) is a question whose answer is yes or no: decision symbol, diamond; `x < 0 ?` is written inside. (c) gives data to the outside: input/output symbol, parallelogram. (d) calls a separately drawn subprocess: predefined process symbol, a rectangle with vertical lines on its sides.

2. A sequential diagram reads a = 5, then runs the boxes `b = a * 2` , `a = b + a` , `c = a + b` in

order and prints a, b, and c. What are the variables after each box, and what is the output?

Answer

`b = a * 2`: *b = 5 × 2 = 10*; a = 5. `a = b + a`: *a = 10 + 5 = 15*; a's old value 5 is now gone. `c = a + b`: *c = 15 + 10 = 25*. The output is `15 10 25`. When c is computed, the new value of a (15) is used.

3. The decisions in the temperature chain are arranged in reverse order: first `t < 25 ?` (Yes → "Mild"), then `t < 15 ?` (Yes → "Cold"), then `t < 0 ?` (Yes → "Freezing", No → "Hot"). What is printed for t = −5, 10, 20, and 30? Which exits can never run?

Answer

For −5, 10, and 20 the first decision is true and all three print "Mild." For 30 the first decision is false. Execution reaches the second decision; *30 < 15* and *30 < 0* are false and "Hot" is printed. The "Cold" and "Freezing" branches cannot be reached with any input: the second decision is reached only when t ≥ 25, and such a t is less than neither 15 nor 0. The right order is from the narrow condition to the wide one, that is, t < 0, t < 15, t < 25.

4. In a loop started with i = 10, the decision is `i > 0 ?`; on the Yes branch i is printed and `i = i - 3` is done. What is the output? How many times do the body and the decision run, and what is i when the loop ends?

Answer

The condition is true while i is 10, 7, 4, 1, in turn; these values are printed: `10 7 4 1`. After 1 is printed, *i = 1 − 3 = −2* and *−2 > 0* is false. The body runs 4 times and the decision 5 times; the loop is exited with i = −2.

5. In a diagram that finds the product of the numbers from 1 to 5, the accumulator was started with `product = 0`; the body is `product = product * i`. What does the diagram print? What does the corrected diagram print?

Answer

In the first iteration *0 × 1 = 0*, and since every later iteration multiplies by 0, the result is 0. The neutral value for a product is 1; starting with `product = 1`, the iterations give 1, 2, 6, 24, 120 and 120 is printed.

6. The inputs 70, 85, 0, −1 are entered, in order, into the sentinel-controlled grade-average diagram. What do the sum, counter, and average become? What would have happened if 0 had been chosen

as the sentinel instead of −1?

Answer

70, 85, and 0 are real grades: the sum is *70 + 85 + 0 = 155* and the counter is 3. When −1 is read the loop ends. The average is *155 / 3 ≈ 51.67*. Had the sentinel been 0, the third input would have been interpreted as "data finished": the sum would be 155, the counter 2, and the average 77.5. Because 0 can be a real grade, it is not suitable as a sentinel.

7. In a nested loop the outer counter i runs from 1 to 3 and the inner counter j from i to 3. The inner body prints one "*"; at the end of each outer iteration a line break is printed. What is the output, and how many stars are printed in all?

Answer

With i = 1, j = 1, 2, 3: three stars. With i = 2, j = 2, 3: two stars. With i = 3, j = 3: one star. The output is the lines `***`, `**`, `*`, one under another. In total *3 + 2 + 1 = 6* stars are printed. Because the start of the inner loop depends on the outer counter, the inner body runs 6 times, not 3 × 3 = 9.

8. The body of a loop that starts with count = 10 prints the counter and increases it by 1; the condition is `count < 5`. What is printed when the loop is drawn as a pre-test (`while`) and as a post-test (`do-while`) loop?

Answer

In the pre-test loop the condition is asked before the body: *10 < 5* is false and the body never runs; nothing is printed. In the post-test loop the body runs first: 10 is printed and count becomes 11. Then *11 < 5* is asked, it is false, and the loop ends. The output is just 10. The two loops behave differently when the condition is false at the first try.

9. Euclid's algorithm is executed for (84, 36). What are a, b, and k on each iteration? How many iterations does the loop run, how many times is the decision asked, and what is printed?

Answer

Initially: a = 84, b = 36; *36 != 0* is true. Iteration 1: *k = 84 % 36 = 12* (84 = 2 × 36 + 12); a = 36, b = 12. Iteration 2: *k = 36 % 12 = 0*; a = 12, b = 0. *0 != 0* is false; the loop ends. 2 iterations, 3 decisions; 12 is printed. Check: *84 = 7 × 12*, *36 = 3 × 12*, and 7 and 3 have no common divisor greater than 1.

4.2.12 Further reading

- C. Böhm, G. Jacopini, “Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules,” *Communications of the ACM*, 9(5), 1966: the original paper showing that every flowchart has a structured equivalent made of sequence, selection, and repetition.
- E. W. Dijkstra, “Go To Statement Considered Harmful,” *Communications of the ACM*, 11(3), 1968: the two-page letter that began the structured-programming debate, arguing why statements that jump at random make programs incomprehensible.
- R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), Section 1.3: conditionals and loops in Java, explained with trace tables and many short programs.
- A. B. Downey, C. Mayfield, *Think Java* (2nd ed., O'Reilly, 2019): conditionals, logical operators, and loops explained in plain language for beginners; the online edition is free.

4.2.13 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. P. Deitel, H. Deitel, *Java: How to Program, Early Objects*, 11th ed., Pearson, 2018.
3. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 1.3 Conditionals and Loops.
4. Oracle, *The Java Tutorials: Control Flow Statements*.
5. A. B. Downey, C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019.

PART II · CHAPTER 4.3

Introduction to Programming

However clearly an algorithm is written on paper, a processor cannot run it directly. The only language a processor understands is machine language, made of zeros and ones. A programming language sits between the two worlds: people can read it, and a translation tool can turn it into machine language. This note first explains what a programming language is, how source code becomes machine code, and how the compiler, the interpreter, and the virtual machine share that work. It then classifies languages by level, type, paradigm, and typing discipline. The last sections show how a problem becomes a program and what stages software development goes through. The basic constructs (variables, decisions, loops, functions, and arrays) and the ways of finding bugs are illustrated with step-by-step trace tables. The code examples are in Java, Python, and C; the outputs of the Java and Python examples come from real runs.

LEARNING OUTCOMES

Concepts
Define a programming language, its syntax, and its semantics; distinguish source code, assembly, and machine language.

Execution
Explain compilation, interpretation, and hybrid execution; place bytecode, the virtual machine, and JIT compilation; list the phases of a compiler and the jobs of the linker and the loader; decide whether an error is a compile-time error, a runtime error, or a logic error.

Languages
Classify languages by level, type, and paradigm; explain when static and dynamic typing catch an error; translate pseudocode into Java and Python.

Problem solving and the software process
Define a problem by its input, output, and constraints and get from the algorithm to a program; recognize the stages of software development and the development tools.

Basic constructs and debugging
Use variables, data types, operator precedence, decisions, loops, functions, and arrays; trace a program step by step with a trace table to find its bug.

4.3.1 Basic concepts

A recipe tells you, step by step, what to do in the kitchen. A person reading it fills in a vague phrase such as "a pinch of salt" from experience. A computer has no such experience. Every step it is given must have exactly one meaning, and that meaning must be defined precisely in advance. Programming languages were born from this need.

4.3.1.1 What is a programming language?

Everyday language is flexible. The sentence "See you tomorrow" does not say at what time; the listener completes the meaning from context. A machine cannot be expected to infer meaning from context. To make a machine do work, we need a language in which every statement has a single interpretation and whose rules are written down beforehand. Such languages are called formal languages.

Definition 4.3.1

A **programming language** is a formal language that lets people express computational steps to a computer. A collection of commands written in such a language is called a **program**.

A programming language is the tool for writing an algorithm in a form the computer can execute. An algorithm is an idea: "add a and b, then print the result." That idea can be written in Java, in Python, or in C. The notations differ, but all of them describe the same steps and produce the same output for the same input. Figure 4.3.1 shows this for a = 3 and b = 5: all three languages print 8.

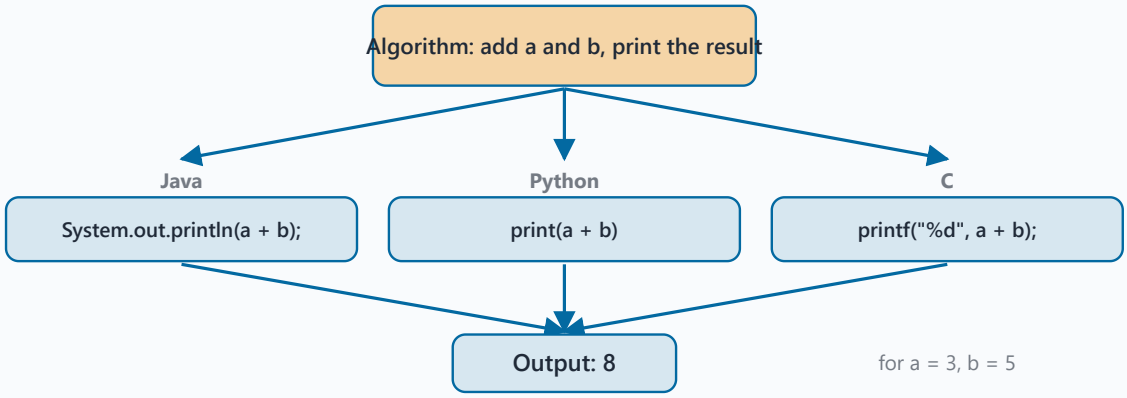


Figure 4.3.1: The same algorithm (add a and b, print the result) written in Java, Python, and C. The syntax rules differ; for a = 3 and b = 5 all three print 8.

The lines in the figure are only the addition part of complete programs. For a program to run, other lines are needed around them. Below is the complete version of the same task in Java and in C. The Java program was compiled and run with JDK 21; the output of the C program was determined by tracing it line by line.

Example 4.3.1

How is a complete program that adds a = 3 and b = 5 and prints the result written in Java and in C?

Java program:

```
public class Sum {
    public static void main(String[] args) {
        int a = 3;
        int b = 5;
        System.out.println(a + b);
    }
}
```

C program:

```
#include <stdio.h>

int main(void) {
    int a = 3;
    int b = 5;
    printf("%d\n", a + b);
    return 0;
}
```

Both programs produce the same output:

```
8
```

In Java every program is written inside a `class`, and execution starts at the `main` method. In C, execution starts at the `main` function; to print to the screen, `printf` from the `stdio.h` library is used. In `printf`, `%d` means "an integer goes here" and `\n` is the end of a line. In Python the same job takes one line: `print(a + b)`. The details differ, but the three steps of the algorithm (set up a, set up b, print the sum) appear in the same order in all three programs.

That the same problem can be solved in different languages raises a question of choice: which language? The answer depends on the needs of the problem and on the environment in which the program will run. An app for a phone, a calculation on a server, and the control software of a washing machine may call for different languages.

Common mistake

Thinking that programming means memorizing a language. The real work of writing a program is building the algorithm: understanding the problem, finding the steps, and making sure the steps run in the right order and under the right conditions. The language is the tool for writing those steps down. A well-built algorithm can be moved from one language to another in a few hours; no language can rescue a badly built one.

4.3.1.2 Syntax and semantics

In English, the sentence "Ali went to school" is well formed. "Went school to Ali" can still be understood but breaks the rules. "The school went to Ali" follows the rules but makes no sense. Programming languages also define these two dimensions separately: whether a piece of text follows the rules, and what a well-formed piece of text does.

Definition 4.3.2

The **syntax** of a programming language is the set of rules that determines which constructs may be written in which order. **Semantics** determines what every well-formed construct does when it is executed.

In Java, ending every statement with a semicolon is a syntax rule. If the line `int b = 5` has no semicolon at the end, the text breaks the rule and the program cannot be translated. That the expression `a + b` computes the sum of a and b is a semantic rule. Python has different syntax:

statements need no semicolon, but the indentation of blocks is part of the rule.

A program with correct syntax does not necessarily give the desired result. A line that divides by 4 instead of 3 when computing the average of three grades follows the rules perfectly. The translator accepts the line without complaint, the program runs, and it prints a wrong number. The text is valid, but what it says is not what the programmer wanted. Such errors are called logic errors.

Example 4.3.2

In what way is each of the following three Java lines problematic: syntax or semantics?

1. `int b = 5` (no semicolon at the end of the line)
2. `double avg = (g1 + g2 + g3) / 4;` (the average of three grades is wanted)
3. `System.out.println("Sum: " + a + b);` (for a = 3, b = 5, "Sum: 8" is wanted)

The first line is a syntax error; the compiler stops with the message "';' expected". The second line follows the rules but divides by the wrong divisor: its meaning does not match the programmer's intent. The third line also follows the rules, but in Java `+` is evaluated from left to right: first "Sum: " is joined with 3, then 5 is appended, and the screen shows "Sum: 35". To get the intended result, the addition must be parenthesized: `"Sum: " + (a + b)`. Only someone who knows the semantic rules of the language can spot the problem in the last two lines.

4.3.1.3 Source code, assembly, and machine language

The text a programmer writes lives in a text file. This file can be opened in Notepad, read, and changed. A processor, however, cannot read this text. The commands a processor understands are encoded as binary numbers, and every processor family has its own instruction set.

Definition 4.3.3

Source code is the text form of a program that a developer can read and change. **Machine language** consists of the binary instructions the processor runs directly; a program made of these instructions is called **machine code**. **Assembly language** is the human-readable form of the machine instructions, written with symbolic names.

Figure 4.3.2 shows how a single line passes through these three forms. The compiler translates the source line `sum = a + b;` into three processor instructions. The first loads the value of a from memory into a processor register (R1). The second adds b to R1. The third writes the result in R1 to the location of sum in memory. Assembly writes these instructions with short names such as LOAD, ADD, and STORE. A tool called an assembler turns these names into binary codes. The binary codes in the figure are only an example; real encodings vary from processor to processor.

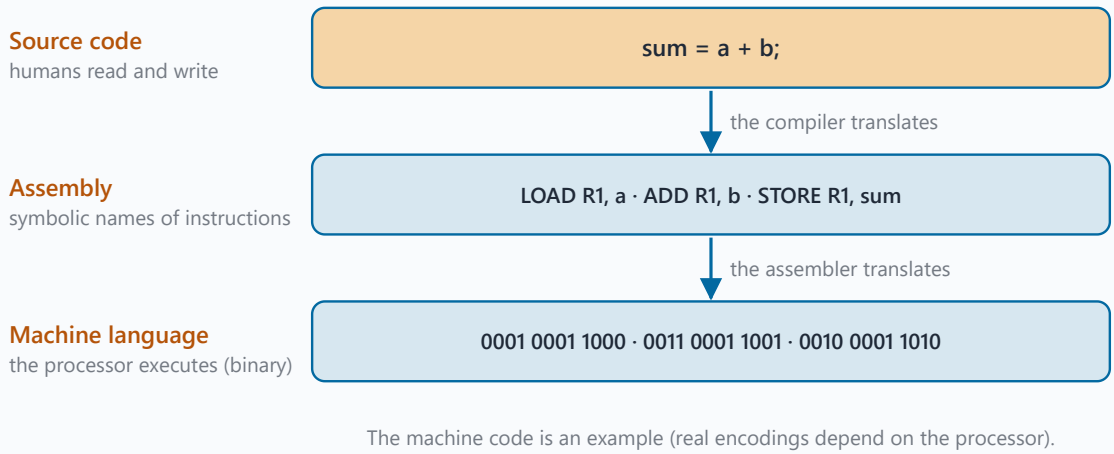


Figure 4.3.2: From source code to machine language. The compiler translates one line into several assembly instructions, and the assembler translates those instructions into binary machine code.

Two consequences follow from this translation. First, a programming language and a processor's language are not the same thing. Java or Python is not the language of any processor; for text written in these languages to reach the processor, a translation is needed. The programs that do this translation are generally called **translators**: compilers and interpreters. Second, one high-level line often corresponds to several machine instructions. The programmer says "add and store"; the translator spells out the load, add, and write steps.

Common mistake

Thinking that the processor "understands" Java or Python source code. The processor runs only its own machine language. That is why double-clicking a `.java` or `.py` file does not run it. The file must first pass through a compiler or an interpreter. In Java this job is split in two: javac translates the source code into bytecode, and the JVM runs the bytecode.

4.3.2 Running a program

After source code is written, several things happen before a result appears on the screen. The text passes through a translator, the translated code runs on the processor or on a virtual machine, and the running program reads its input and produces its output. A different kind of error can arise at each stop along this path.

4.3.2.1 From source code to output

Figure 4.3.3 shows the path of a Java program in five stops. The programmer writes the source code in a text file named `Sum.java`. The computer cannot run this text as it is. At the translation stop, the compiler reads the text and checks whether it obeys the rules of the language. If there are no errors, it

produces an output file. In languages such as C, this file is machine code directly. In Java it is an intermediate-code file named `Sum.class`. At the execution stop the instructions are carried out on the processor; in Java this is done through the Java Virtual Machine (JVM). At the last stop the program produces its output: 8 for a = 3 and b = 5.

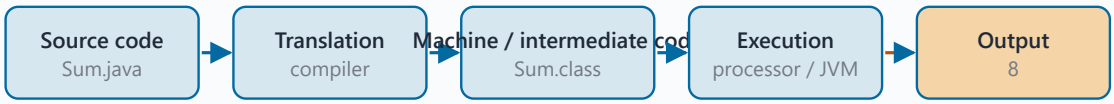


Figure 4.3.3: The path from source code to output. The source code is translated, the translation yields machine code or intermediate code, that code runs on the processor or the JVM, and output is produced.

In Java this path is walked with two commands at the command line. `javac Sum.java` runs the compiler and produces the file `Sum.class`. `java Sum` starts the JVM and runs the program in that file. It matters that the two steps are separate: compilation is done once, and the resulting file can be run as many times as you like. If the source code is changed, compilation must be repeated; otherwise the JVM keeps running the old `.class` file.

Common mistake

Fixing the source code, running `java Sum` without recompiling, and concluding that "the fix did not work." The JVM runs the compiled `.class` file, not the source code. Every time the source code changes, `javac` must be run again. IDEs do this step automatically when you save or press the run button; for that reason the step is often overlooked.

4.3.2.2 Kinds of errors

Every stop on the path exposes its own kind of error. Knowing at which stop an error shows up also tells us where to look for it.

Definition 4.3.4

A **compile error** is source code that does not obey the rules of the language; it is found during translation and the program never runs. Syntax errors and type errors are of this kind. A **runtime error** is an error that arises while the program is running and stops it. A **logic error** is when the program runs without any error message but produces a wrong result.

It is easy to see the three kinds in three small Java programs. In the first program, the semicolon at the end of the second assignment line has been forgotten:

```
public class MissingSemicolon {
    public static void main(String[] args) {
        int a = 3;
        int b = 5
        System.out.println(a + b);
    }
}
```

The compiler refuses to translate the program and reports the location of the error with a line number:

```
MissingSemicolon.java:4: error: ';' expected
        int b = 5
                ^
1 error
```

The message gives the file name, the line number (4), and the expected token (`;`). The `^` below it shows where the compiler expected the semicolon. In this case no `.class` file is produced and no line of the program runs; "8" never appears on the screen. Figure 4.3.4 shows this situation with the red mark at the translation stop of the pipeline.

In the second program the syntax is flawless, but line 5 contains a division by zero:

```
public class DivideByZero {
    public static void main(String[] args) {
        System.out.println("Hello");
        int x = 10;
        System.out.println(x / 0);
        System.out.println("End");
    }
}
```

Compilation finishes successfully. When the program runs, "Hello" is printed first, then the JVM tries to evaluate `x / 0` and stops the program:

```
Hello
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideByZero.main(DivideByZero.java:5)
```

In the output, `ArithmeticException: / by zero` names the division-by-zero error, and `DivideByZero.java:5` gives the line of the error in the source file. The "End" line never runs. The compiler cannot catch this error, because the value of x and the division are computed only at run

time. If the divisor were user input, it would be impossible to know at compile time whether it would be zero.

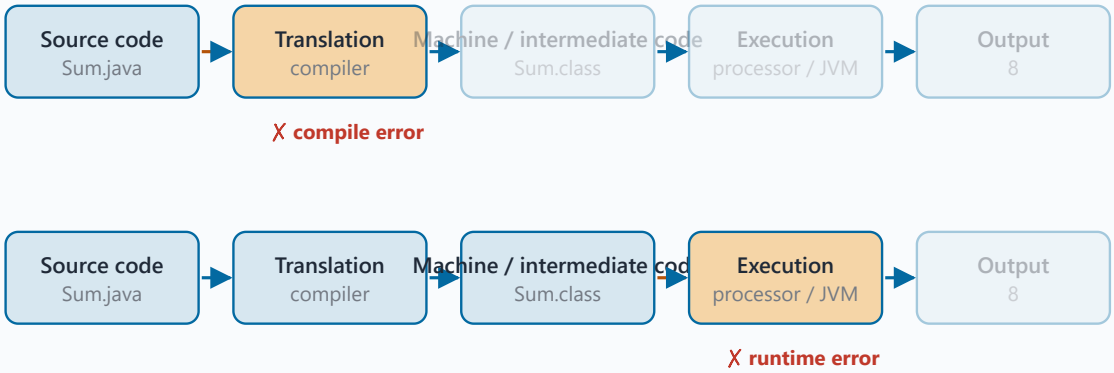


Figure 4.3.4: Where errors occur on the pipeline. Top: a syntax error is caught at the translation stop and the program never runs. Bottom: the division by zero appears at the execution stop; the lines before it have already run.

The third program computes the average of three grades, but a parenthesis is missing:

```
public class AverageBug {
    public static void main(String[] args) {
        int g1 = 70, g2 = 85, g3 = 90;
        System.out.println(g1 + g2 + g3 / 3);
        System.out.println((g1 + g2 + g3) / 3);
        System.out.println((g1 + g2 + g3) / 3.0);
    }
}
```

```
185
81
81.66666666666667
```

The program compiles without trouble, runs without any error message, and prints 185 on the first line. The expected result is a number between 81 and 82. Because division happens before addition, only g3 is divided by three: *70 + 85 + 90 / 3 = 70 + 85 + 30 = 185*. This is a logic error. The second line adds the parentheses but is still not quite right. In Java the quotient of two integers is an integer; the fractional part of *245 / 3* is discarded and the result is 81. The third line divides by `3.0`; the computation is done in floating point and the correct value, about 81.67, comes out. This is what logic errors have in common: the program "works," and only someone who compares the expected result with the actual one notices the problem.

Kind of error	When does it appear?	Who reports it?	Example
Compile (syntax, type)	During translation; the program never runs	The compiler, with a line number	A forgotten `;`, assigning text to an int
Runtime	During execution, when the faulty line is reached	The JVM or the operating system; the program stops	Division by zero, going outside an array
Logic	Never shows up by itself	Nobody; found only by testing	A missing parenthesis, a wrong divisor

Table 4.3.1: Comparison of the three kinds of errors

Example 4.3.3

What kind of error is each of the following? (a) In Java, the line `System.out.println("Hello")` has no semicolon at the end. (b) When the user enters 0 for the number of students, the program stops while computing the average. (c) When computing a class average, the total is divided by n + 1 instead of the number of students n.

1. (a) A compile error. The semicolon is a syntax rule; the compiler reports it with a line number and no program is produced.
2. (b) A runtime error. The code obeys the rules and compiles; but the moment the divisor is 0, execution stops. The same program runs fine with non-zero inputs, so the error shows up only with certain inputs.
3. (c) A logic error. The program runs with every input and prints a number; the number it prints is always smaller than it should be. For grades 60, 70, and 80, for example, it prints *210 / 4 = 52.5*; the correct value is *210 / 3 = 70*.

Common mistake

Counting the absence of an error message as proof that a program is correct. The compiler only checks whether the rules were followed; it cannot know whether the program does the intended job. Every program must be tried on a few inputs whose results have been computed by hand.

4.3.2.3 Compiled, interpreted, and hybrid languages

There are three common ways to run source code. The difference lies in when the translation is done and how much of it is done.

Definition 4.3.5

A **compiler** is a program that translates the whole program into target code before it runs; this process is called **compilation**. An **interpreter** is a program that runs a program by evaluating it command by command at run time, without first translating it into an executable file; this process is called **interpretation**. In **hybrid** execution, the source code is first compiled to an intermediate code, and the intermediate code is executed on a virtual machine.

Approach	How it works	Example languages
Compiled	The program is translated beforehand into target machine code	C, C++, Rust
Interpreted	Commands are evaluated at run time	Python, JavaScript
Hybrid	Intermediate code is executed on a virtual machine	Java, C#

Table 4.3.2: Execution approaches

Figure 4.3.5 shows the three paths side by side. On the compiled path, the file `source.c` goes through the compiler and an executable such as `program.exe` comes out. This file runs on the processor directly; at run time the compiler is no longer needed. On the interpreted path, the file `source.py` is handed to the interpreter every time it runs. The interpreter processes the commands in order and has the processor carry them out. There is no lasting machine-code file in between. On the hybrid path, `Sum.java` is first compiled with `javac` into `Sum.class`. The JVM runs this file, and the JVM itself is a program running on the processor.

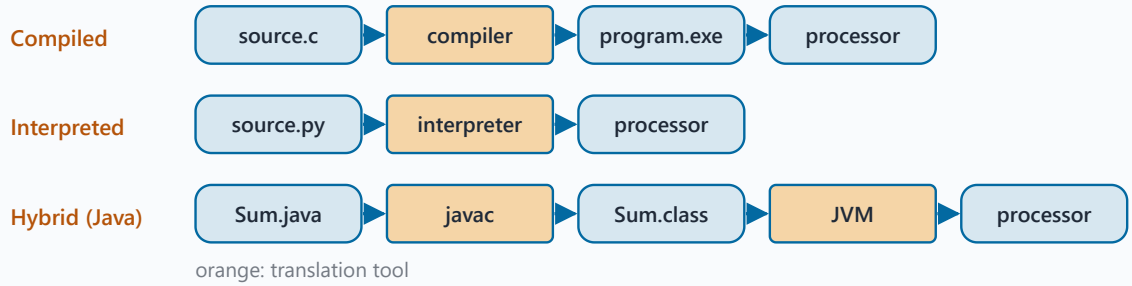


Figure 4.3.5: The compiled, interpreted, and hybrid execution paths. The orange boxes are translation tools: the compiler, the interpreter, javac, and the JVM.

Which path is chosen also determines how the program is distributed. A compiled C program is valid only for the processor and operating system it was compiled for. An `.exe` file compiled for Windows does not run on Linux or on a computer with a different processor. An interpreted Python program is distributed as source code; it runs on every computer that has the Python interpreter installed. Java's `.class` file, on the other hand, is processor-independent: the same file runs on every platform that has a JVM. Java's well-known slogan "write once, run anywhere" comes from this property.

Common mistake

Putting Java in the same group as C as a "compiled language." Java source code is compiled, but what the compiler produces is not machine code but bytecode, which the JVM understands. This code does not run directly on the processor; a JVM is needed to run it. Placing Java in the hybrid class describes this two-stage structure correctly.

4.3.2.4 The compiler and the interpreter step by step

The difference between a compiler and an interpreter is best seen on the same small program. The third line of the following three-line pseudocode contains a division by zero:

Line	Pseudocode
1	`print "Hello"`
2	`x ← 10`
3	`print x / 0`

In pseudocode, assignment is written with `←`: `x ← 10` means "give x the value 10." This keeps assignment from being confused with an equality comparison. Let the program be processed first by a compiler and then by a simple interpreter that runs line by line. What happens at each step is listed in Table 4.3.3.

Step	Compiler	Simple interpreter
1	Lines 1, 2, and 3 are read and translated from start to end; no line has run yet	Line 1 is translated and runs immediately: `Hello`
2	The division is well formed; translation succeeds and an executable is produced	Line 2 is translated and runs: x = 10; line 3 has not been read yet
3	Execution begins; line 1: `Hello`	Line 3 is translated, and an error occurs while it runs: division by zero
4	Line 2: x = 10; line 3: runtime error	—
Screen	`Hello`, then the error	`Hello`, then the error

Table 4.3.3: Processing the same program with a compiler and with an interpreter

In this program both paths end with the same screen: "Hello" followed by an error. The difference lies not in the kind of error but in the order of the work. The compiler translates the whole text first, and execution comes later as a separate stage. The interpreter interleaves translating and running line by line. Because division by zero is a runtime error, in both paths it shows up only when the third line runs. Figure 4.3.6 shows the final state of the two paths.

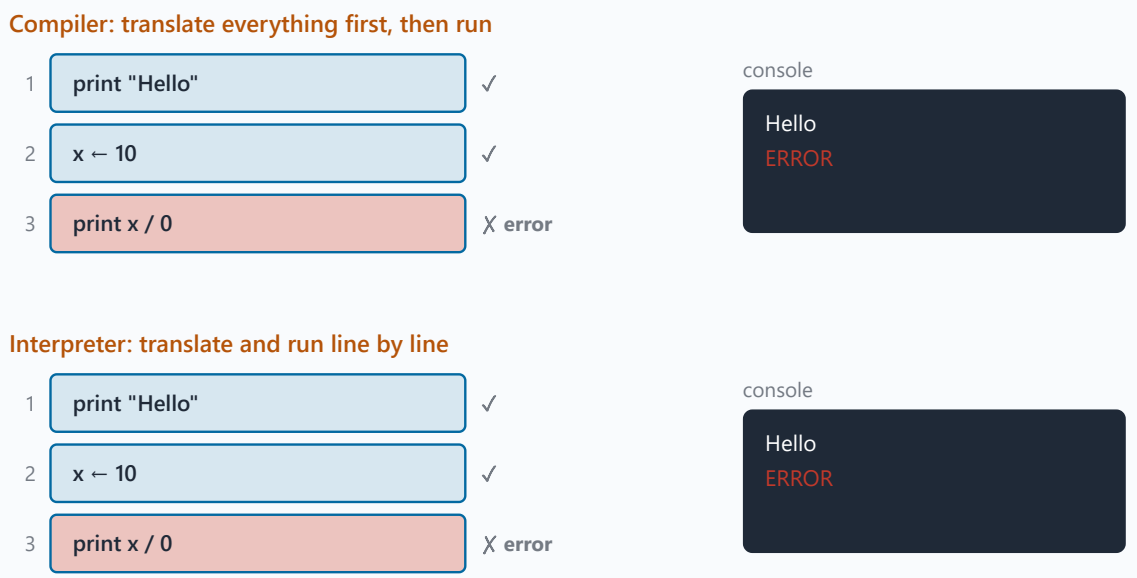


Figure 4.3.6: The final state of the three-line program. Top, the compiler: all three lines were translated first (✓), and line 3 failed at run time. Bottom, the interpreter: each line was translated and run immediately; the error comes when line 3 is reached.

When the error is a syntax error, the two paths part ways. Suppose a parenthesis is left unclosed on the third line: `print (x + 2`. The compiler stumbles on this line during translation and produces no executable. The screen shows only the message "line 3: syntax error"; not even "Hello" is printed. A simple line-by-line interpreter, however, runs line 1 and prints "Hello," assigns 10 to x on line 2, and notices the error only when it reaches line 3.

Modern interpreters such as those for Python and JavaScript depart a little from this simple model: before they start running, they parse the whole file. They therefore report syntax errors at the very start. The following Python file shows this:

```
print("Hello")
x = 10
print((x + 2)
print("End")
```

```
  File "syntax_error.py", line 3
    print((x + 2)
          ^
SyntaxError: '(' was never closed
```

"Hello" does not appear on the screen. Before running the first line, Python read the whole file and found the unclosed parenthesis on line 3. When the same file has a division by zero instead of a syntax error, the situation changes:

```
print("Hello")
x = 10
print(x / 0)
print("End")
```

```
Hello
Traceback (most recent call last):
  File "zero_division.py", line 3, in <module>
    print(x / 0)
          ~~^~~
ZeroDivisionError: division by zero
```

This time "Hello" is printed, because a division by zero is detected only when the third line runs. Python's message resembles Java's: the file name, the line number, the kind of error (`ZeroDivisionError`), and a short description.

Example 4.3.4

A four-line program has a syntax error on line 2 and a division by zero on line 4. What appears on the screen if the program is run (a) with a compiler, (b) with a simple line-by-line interpreter, (c) with Python? Line 1 prints "Start" and line 3 prints "Middle".

1. (a) The compiler first translates all four lines. It stops at the syntax error on line 2 and produces no executable. The screen shows only a compile error message for line 2; "Start" is not printed.
2. (b) The simple interpreter translates and runs line 1: "Start". When it tries to translate line 2, it stops with the syntax error. Lines 3 and 4 are never reached. The screen shows "Start" and the error message.
3. (c) Python first parses the whole file and finds the error on line 2. The screen shows only the `SyntaxError` message; the behavior is the same as in (a). The division by zero on line 4 never shows up in any of the three cases, because the program would have to run that far.

4.3.2.5 Bytecode, the virtual machine, and JIT compilation

Python is called an "interpreted language," but the Python interpreter also contains a compilation step. CPython, the most common implementation of Python, first compiles the source code into an intermediate code called **bytecode**. It then executes this bytecode command by command on a virtual machine inside it. No machine-code file results. The compilation step is hidden from the programmer: the command `python file.py` performs the two stages one after the other.

Definition 4.3.6

Bytecode is an intermediate code made of simple instructions designed for a virtual machine rather than for a particular processor. A **virtual machine** is software that reads and executes these instructions; it imitates the work of a real processor in software.

The bytecode Python produces can be seen with the `dis` module. The following program compiles the line `area = width * height + 2` and prints its bytecode:

```
import dis
dis.dis(compile("area = width * height + 2", "<code>", "exec"))
```

```
  0           RESUME                   0

  1           LOAD_NAME                0 (width)
              LOAD_NAME                1 (height)
              BINARY_OP                5 (*)
              LOAD_CONST               0 (2)
              BINARY_OP                0 (+)
              STORE_NAME               2 (area)
              RETURN_CONST             1 (None)
```

Each line of the output is a virtual machine instruction. `LOAD_NAME` pushes the value of a variable onto the virtual machine's stack. `BINARY_OP 5 (*)` multiplies the top two values on the stack, `LOAD_CONST` pushes the constant 2, and `BINARY_OP 0 (+)` adds. `STORE_NAME` binds the result to the name area. Instruction names may change slightly between Python versions; Python 3.13 was used here. The order, however, is the same in every version: first width and height are multiplied, then 2 is added. This order comes from operator precedence.

Executing bytecode instruction by instruction is slower than running machine code directly. For every instruction the virtual machine has to answer the question "what does this instruction mean?" The way to reduce this slowness is **JIT compilation**.

Definition 4.3.7

A **just-in-time (JIT) compiler** is a compiler that translates frequently executed pieces of code into machine code while the program is running. The translated piece is not interpreted on later runs; it runs directly on the processor.

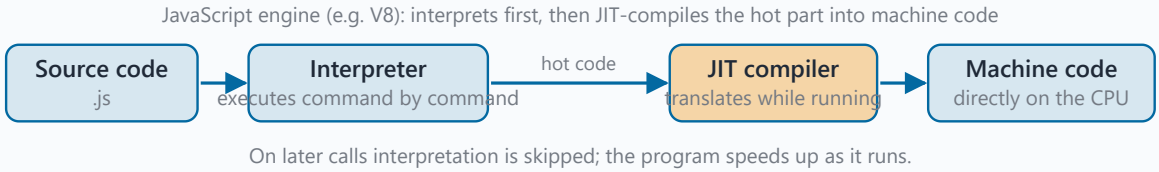


Figure 4.3.7: JIT compilation. A JavaScript engine (e.g. V8) first interprets the code; it translates the frequently executed part into machine code at run time. On later calls interpretation is skipped and the program speeds up as it runs.

Figure 4.3.7 shows this idea using JavaScript. The V8 engine in the Chrome browser first interprets the code. When it sees that a function has been called thousands of times, it compiles it to machine code. The JVM follows the same path: bytecode is interpreted at first, and frequently run methods are translated into machine code by the JIT. For this reason, long-running Java programs can reach speeds close to those of compiled C programs.

These examples bring out an important point: "compiled" or "interpreted" is a property not of a language but of an implementation of that language. Implementations of Python that compile to machine code have been written, and so have interpreters for C. Most real systems combine the two methods. Calling a language "interpreted" is a short way of saying how its most common implementation works.

Common mistake

Saying "Python is an interpreted language, so no compilation takes place." CPython compiles the source code to bytecode on every run. It stores the bytecode of imported modules as `.pyc` files in the `__pycache__` folder and reuses these files on later runs if the source has not changed. What makes Python interpreted is that the bytecode is executed command by command on a virtual machine, not as machine code.

4.3.2.6 The phases of a compiler

A compiler does not work in one big step. It splits its job into five consecutive phases; the output of each phase is the input of the next. Following the phases on a single line also explains which error the compiler catches and where. The line to follow is this:

```
area = width * height + 2;
```

1. Lexical analysis. The compiler sees the line as a sequence of characters: a, r, e, a, space, = ... The first phase splits these characters into meaningful pieces called **tokens**. Whitespace and comments are discarded in this phase. Eight tokens come out of the example line, and the type of each is known (Figure 4.3.8).

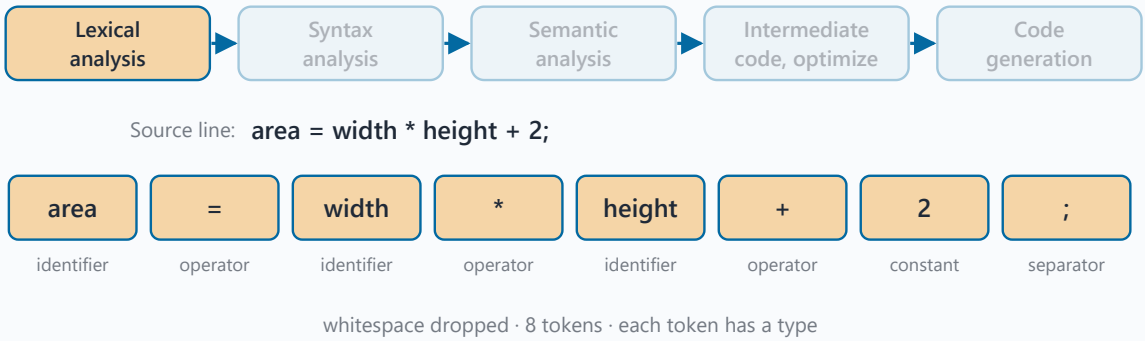


Figure 4.3.8: Lexical analysis. The line `area = width * height + 2;` is split into eight tokens: three identifiers, three operators, one constant, and one separator.

The errors caught in this phase are character sequences that match no token of the language. In Java the `#` character is not an operator. An unclosed string cannot be turned into a token either. The JDK 21 compiler reports these two cases as follows:

```
IllegalCharacter.java:4: error: illegal character: '#'
        int area = width # height;
                         ^
IllegalCharacter.java:4: error: not a statement
        int area = width # height;
                           ^
2 errors
```

```
UnclosedString.java:3: error: unclosed string literal
        System.out.println("Hello);
                           ^
1 error
```

2. Syntax analysis (parsing). The tokens are arranged into a tree according to the rules of the language; this tree is called a **syntax tree**. Figure 4.3.9 shows the tree of the example line. At the root is the assignment (`=`); its left branch is area and its right branch is the expression `width * height + 2` . The multiplication is deeper in the tree, so it is evaluated first. Operator precedence is built into the shape of this tree in exactly this way.

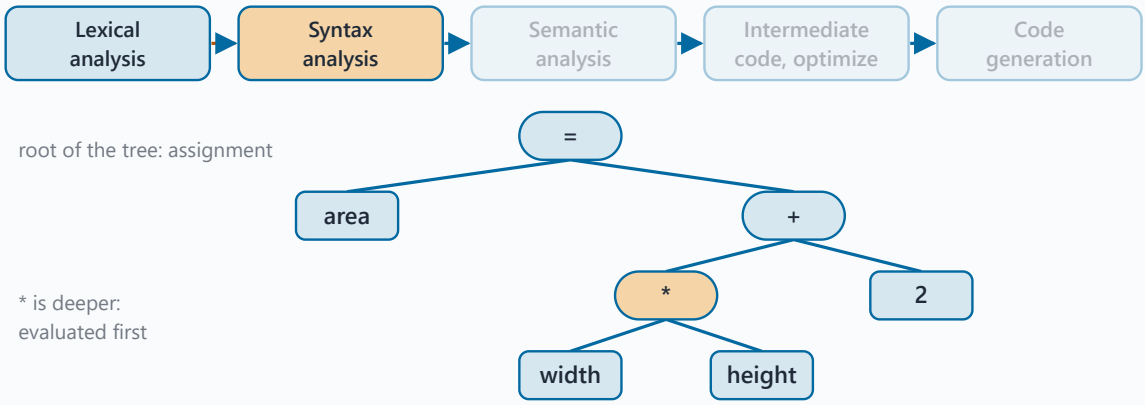


Figure 4.3.9: Syntax analysis. The tokens are arranged into a tree; the assignment is at the root, and the deeper multiplication is evaluated first.

A missing semicolon, an unclosed parenthesis, and tokens written in the wrong order are caught in this phase. For example, in the line `int 2x = 5;` lexical analysis sees no problem: 2 is a number and x is an identifier. But a number following the `int` token breaks the rules. The compiler therefore reports the error in the syntax phase:

```
InvalidToken.java:3: error: not a statement
        int 2x = 5;
        ^
InvalidToken.java:3: error: ';' expected
        int 2x = 5;
           ^
2 errors
```

3. Semantic analysis. A tree with correct syntax is checked for meaning. Every name that is used must have been declared earlier. The types of the operands must also fit together. In the example line, width and height are declared as int; the expression *int × int + int* yields int and can be assigned to area, which has type int (Figure 4.3.10).

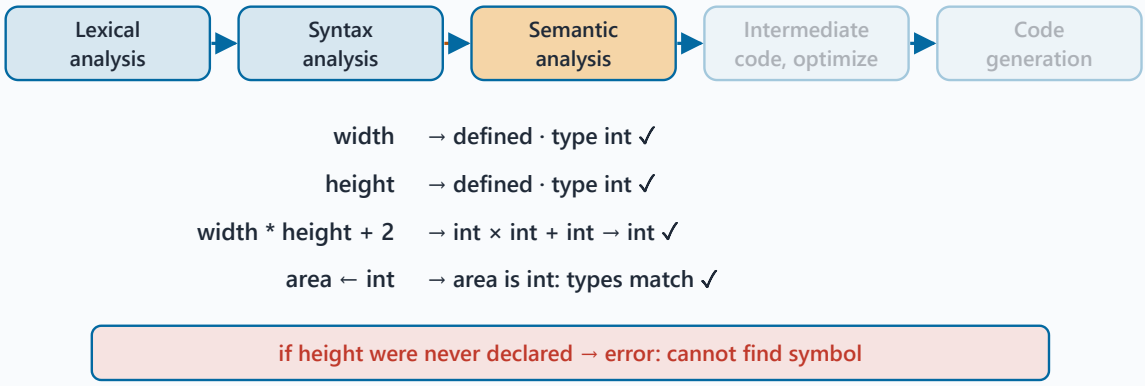


Figure 4.3.10: Semantic analysis. Every name is checked to be defined and the types are checked to be compatible; if height had never been declared, a "cannot find symbol" error would result.

If the variable height is used without being declared, javac gives this message:

```
public class Undeclared {
    public static void main(String[] args) {
        int width = 3;
        int area = width * height + 2;
    }
}
```

```
Undeclared.java:4: error: cannot find symbol
        int area = width * height + 2;
                           ^
  symbol:   variable height
  location: class Undeclared
1 error
```

The message says that the symbol that could not be found is a variable and that its name is height. Type mismatches are also the job of this phase. Assigning text to an int variable is flawless as far as syntax goes, but invalid as far as meaning goes:

```
public class TypeMismatch {
    public static void main(String[] args) {
        int x = 5;
        x = "five";
    }
}
```

```
TypeMismatch.java:4: error: incompatible types: String cannot be converted to int
        x = "five";
            ^
1 error
```

The message means "type String cannot be converted to int." The compiler points at the value `"five"` on line 4 before the program ever runs.

4. Intermediate code generation and optimization. The checked tree is turned into simple instructions, each of which performs a single operation. This form is called three-address code, because each instruction mentions at most three names:

```
t1 = width * height
t2 = t1 + 2
area = t2
```

Here t1 and t2 are temporary names the compiler invents. **Code optimization** is performed on this form: unnecessary operations are dropped, and operations whose results are known at compile time are computed in advance. If width = 3 and height = 4 are constants that never change, the computation *3 × 4 + 2 = 14* is done before the program runs, and only `area = 14` is left in the code. This optimization is called **constant folding** (Figure 4.3.11).

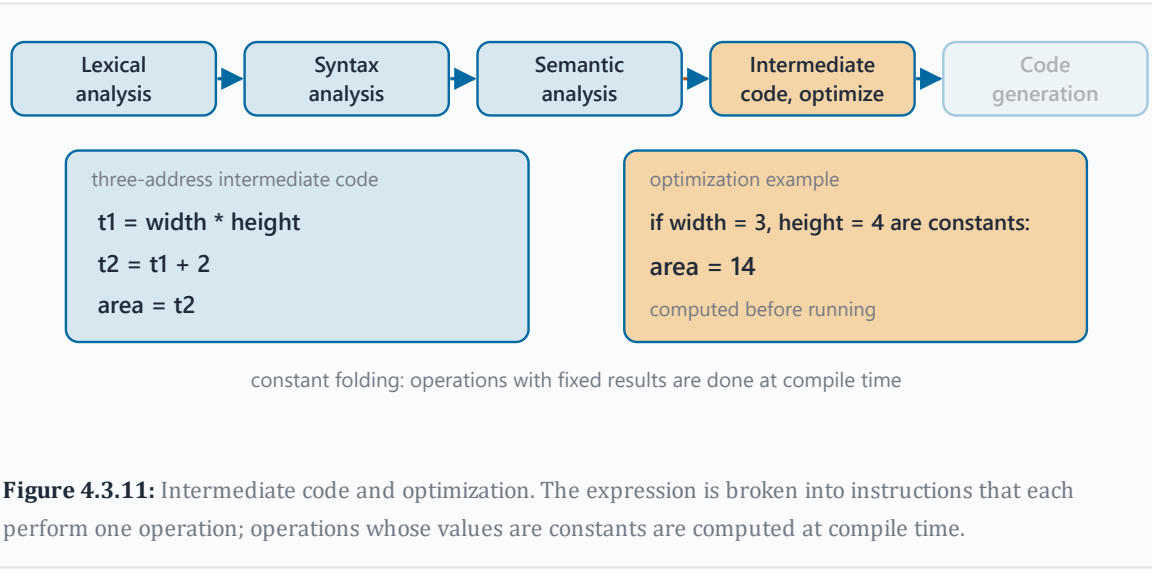


Figure 4.3.11: Intermediate code and optimization. The expression is broken into instructions that each perform one operation; operations whose values are constants are computed at compile time.

5. Code generation. In the last phase the instructions of the target processor are selected and translated into binary code. For the example line, a processor would get LOAD, MUL, ADD, and STORE instructions (Figure 4.3.12). The target of the Java compiler is not a real processor but the JVM. For this reason javac produces bytecode instead of machine instructions.

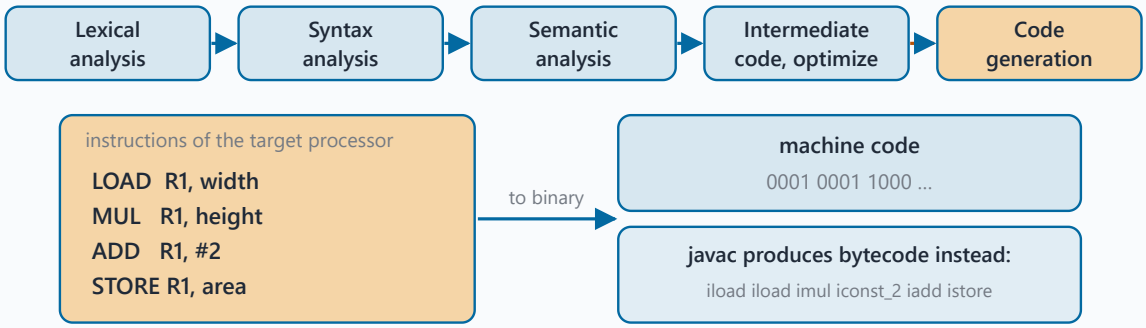


Figure 4.3.12: Code generation. The instructions of the target processor are selected and translated to binary; javac produces bytecode for the JVM instead.

Example 4.3.5

The bytecode javac produces can be read with the `javap -c` command that comes with the JDK. The only difference between the following two programs is that in the second, width and height are declared `final` (unchangeable). How do the bytecodes of the two programs differ?

```
public class AreaExpr {
    public static void main(String[] args) {
        int width = 3;
        int height = 4;
        int area = width * height + 2;
        System.out.println(area);
    }
}
```

```
public static void main(java.lang.String[]);
    Code:
       0: iconst_3
       1: istore_1
       2: iconst_4
       3: istore_2
       4: iload_1
       5: iload_2
       6: imul
       7: iconst_2
       8: iadd
       9: istore_3
      10: getstatic     #7                  // Field
java/lang/System.out:Ljava/io/PrintStream;
      13: iload_3
      14: invokevirtual #13                 // Method java/io/PrintStream.println:
(I)V
      17: return
}
```

In the bytecode of the first program, `iconst_3` and `istore_1` store the value 3 in local variable number 1 (width). `iconst_4` and `istore_2` store 4 in height. The next five instructions (`iload_1` , `iload_2` , `imul` , `iconst_2` , `iadd`) compute the expression, and `istore_3` stores the result in area. This sequence is the same as the "iload iload imul iconst_2 iadd istore" sequence in the slides.

```
public class ConstantFolding {
    public static void main(String[] args) {
        final int width = 3;
        final int height = 4;
        int area = width * height + 2;
        System.out.println(area);
    }
}
```

```
public static void main(java.lang.String[]);
    Code:
       0: bipush        14
       2: istore_1
       3: getstatic     #7                  // Field
java/lang/System.out:Ljava/io/PrintStream;
       6: iload_1
       7: invokevirtual #13                 // Method java/io/PrintStream.println:
(I)V
      10: return
}
```

The second program has no multiplication or addition instructions at all. The instruction `bipush 14` pushes the value 14 directly onto the stack. Because width and height are `final`, the compiler knows their values will not change and performs *3 × 4 + 2* at compile time. This is a real example of constant folding. Both programs print 14 when run.

The first three phases look for errors and report any they find with a line number. The last two phases produce the code that will run. Value-dependent errors such as division by zero show up in no phase, because the compiler does not run the program; it only looks at its text.

Phase	Input → output	Error caught (javac message)
Lexical analysis	characters → tokens	Illegal character (`illegal character`), unclosed string (`unclosed string literal`)
Syntax analysis	tokens → syntax tree	Missing `;` (`';' expected`), unclosed parenthesis
Semantic analysis	tree → checked tree	Undeclared name (`cannot find symbol`), type mismatch (`incompatible types`)
Intermediate code and optimization	tree → simple instructions	—
Code generation	simple instructions → machine code or bytecode	—

Table 4.3.4: The phases of a compiler and the errors they catch

Common mistake

Always assuming that the line number the compiler reports is the error itself. The compiler reports the place where it noticed the error; this is sometimes one line after the real error. When the semicolon at the end of a line is forgotten, some compilers see the error at the start of the next line. If the message points at one line, look at the previous line too.

4.3.2.7 The linker and the loader

Real programs do not consist of a single file. In a calculator program, the user interface may live in one file and the calculation functions in another. For the program to print to the screen, open a file, or take a square root, ready-made library code is also used. The compiler translates each source file separately. Combining these pieces into one program and loading it into memory to run is the job of two separate tools.

Definition 4.3.8

An **object file** is a machine-code file, produced by compiling a single source file, that cannot run on its own (`.o` or `.obj`). The **linker** is the program that combines object files and the library code in use into a single **executable file**. The **loader** is the part of the operating system that brings the executable file into memory and starts it running.

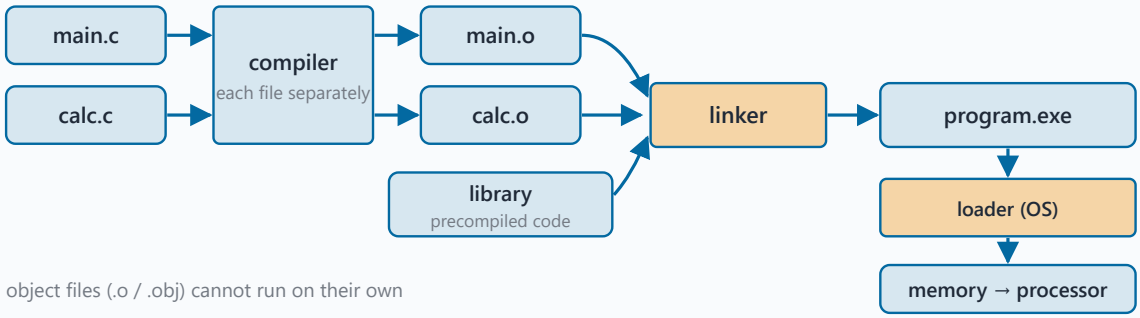


Figure 4.3.13: Two source files are compiled separately; the linker combines the object files and the library, and the loader brings the program into memory.

Figure 4.3.13 shows the path of a C program. `main.c` contains the main program and `calc.c` contains the calculation functions. The compiler translates them into the object files `main.o` and `calc.o` . In `main.o` there is a call to a function in `calc.c` , but the address of that function is not yet known. The compiler only leaves a note: "a function with this name is being called here." The linker resolves these notes. It finds the called function in `calc.o` and ready-made functions such as `printf` in the library. It ties every call to the correct address and finally writes the file `program.exe` .

If a called function cannot be found in any piece, the linker reports a **linking error**. This error differs from a compile error. Each file may have compiled flawlessly on its own; what is missing is the connection between the pieces. For example, if the function `average` called in `main.c` is misspelled `averge` in `calc.c` , the two files compile separately, but the linker cannot find the name `average` anywhere.

When the program is double-clicked or invoked from the command line, the loader steps in. It copies the file from disk into memory. It also loads and links the shared libraries the program needs: `.dll` files on Windows, `.so` files on Linux. Finally it directs the processor to the first instruction of the program. Many programs share a shared library; the same code does not have to be copied into each program separately.

In Java there is no separate linking step. Each class is compiled into its own `.class` file. The JVM's **class loader** finds and loads a class's file at the moment the class is first needed while the program runs. If a class file is missing, the error appears not during compilation or linking but at run time.

Common mistake

Mistaking an object file (`.o` , `.obj`) for an executable program. An object file contains machine code, but calls to other files and libraries are not yet linked. To run, it must go through the linker.

4.3.2.8 Compile or interpret?

The choice between compiling and interpreting is a trade-off. On one side are speed and early error checking; on the other are quick experimentation and flexibility. Table 4.3.5 compares the three approaches on five criteria.

Criterion	Compiled (C)	Interpreted (Python)	Hybrid (Java)
Execution speed	Usually fastest: direct machine code	Slower: commands are decoded while running	Close to compiled, thanks to the JIT
Error reporting	Syntax and type errors before running	Type errors mostly only when that line runs	Most at compile time, the rest at run time
Distribution	A separate file for each operating system and processor	Source code + an interpreter installed on the target	A single `.class` / `.jar` + a JVM on every platform
Development cycle	Recompile after every change	Save and run immediately	Compile, then run
Startup	Starts immediately, no extra software needed	The interpreter must be loaded first	The JVM must be started first

Table 4.3.5: Comparison of compiled, interpreted, and hybrid execution

A game engine or a video encoder cares about every millisecond; compiled C or C++ is preferred for this reason. A data analyst asking different questions of a dataset does not want to wait for compilation after every attempt; Python's "save and run" cycle suits that work. A bank's server software runs for years and is moved to different hardware. Java's portable bytecode and the speed it gains from the JIT fit this need. That both a compiler and an interpreter can be written for the same language is also a result of this trade-off.

Example 4.3.6

A student wants to run the same program on three different computers: a Windows laptop, a Linux server, and a macOS desktop. If the program were written in C, Python, and Java, what would have to be distributed in each case?

1. C: three executables, each compiled separately for its platform. The `.exe` produced on Windows does not run on Linux. The source code can also be distributed; then every computer must have a C compiler.
2. Python: the source file (`.py`). The Python interpreter must be installed on all three computers.

3. Java: a single `.class` or `.jar` file. All three computers must have a JVM; the same file runs unchanged on all three.

4.3.3 Programming languages

The number of programming languages in use today is counted in the hundreds. It is neither possible nor necessary to know them all. Grouping languages by a few criteria makes it easier to place a new language when you meet one. The main criteria are closeness to the hardware, the era of origin, the purpose of use, the way of thinking, and the time of type checking.

4.3.3.1 Language levels

Definition 4.3.9

A **low-level language** is a language whose instructions correspond one-to-one or nearly one-to-one to processor instructions: machine language and assembly. A **high-level language** is a language that lets you write a problem close to the way people express it and hides hardware details: C, Java, Python.

Figure 4.3.14 arranges languages on a ladder. At the bottom is machine language: `10110000 01100001`. On Intel-compatible processors, these two bytes are the instruction "load the value 61h (hexadecimal 61, decimal 97) into the AL register." Assembly writes the same instruction as `MOV AL, 61h`. One rung up is C: the line `x = x + 1;` says nothing about registers; it talks about a variable. Java and C# are higher still; they take memory management away from the programmer. Python is at the top: writing `print(x)` needs no class or type declaration.

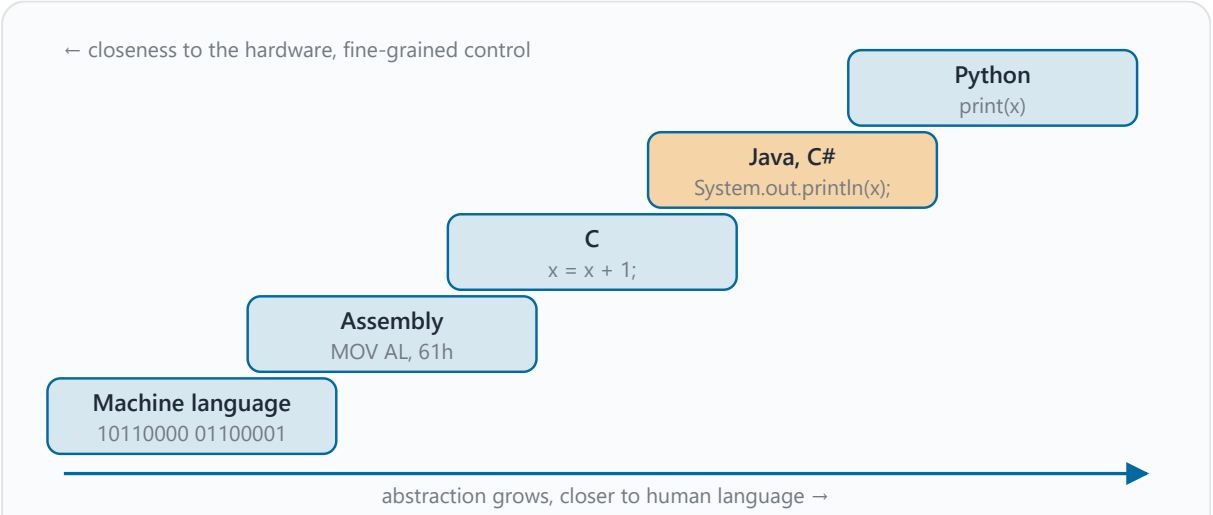


Figure 4.3.14: The ladder of language levels. Going up, abstraction increases and the language gets closer to

human language; going down, closeness to the hardware and fine-grained control increase.

Climbing the ladder brings a gain and a cost. The gain is writing the same job in fewer and clearer lines. Printing a string to the screen takes one line in Python; in assembly it can take dozens. The cost is less control over the hardware. If a driver has to write to a specific memory address at a specific moment, languages on the lower rungs such as C or assembly are needed. So level is not a measure of quality; it describes a language's degree of abstraction. A high-level language is not "better" than a low-level one; it is better suited to other jobs.

Example 4.3.7

What number is the second byte in the machine instruction `10110000 01100001` ? Why is this number written as `61h` in assembly?

1. The second byte is `01100001`. By place values: *0·128 + 1·64 + 1·32 + 0·16 + 0·8 + 0·4 + 0·2 + 1·1 = 64 + 32 + 1 = 97*.
2. The byte splits into two groups of four bits: `0110` = 6 and `0001` = 1. Every four bits make one hexadecimal digit; the number is 61 in base 16. Check: *6 · 16 + 1 = 97*.
3. The trailing h shows that the number is hexadecimal. Assembly programmers like hexadecimal notation because each digit corresponds to exactly four bits, which makes conversion to binary easy. 97 is the code of the lowercase letter "a" in the ASCII table.

4.3.3.2 The evolution of programming languages

The first computers were programmed directly in machine language. Programmers wrote instructions as binary numbers, sometimes flipping switches by hand one at a time. Assembly appeared in the late 1940s: instructions were given memorable names, and assembler programs were written to translate these names into binary. Figure 4.3.15 summarizes the following seventy years on a timeline.

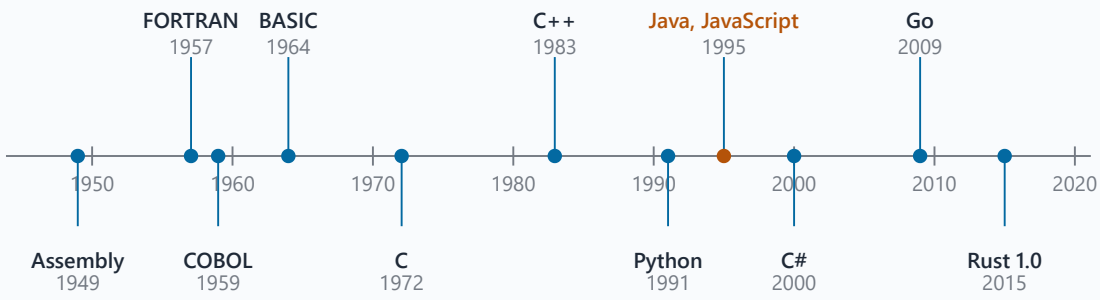


Figure 4.3.15: Timeline of programming languages, 1949–2015. The year under each language is the year its first version or name was announced.

The development can be divided into three periods.

- **1940–1970.** The era of machine language and assembly. In 1957, John Backus's team at IBM developed FORTRAN. FORTRAN was the first widely used high-level language that let scientific formulas be written almost as in mathematics. In 1959, COBOL was designed for business and accounting applications. In 1964, BASIC was developed at Dartmouth College so that students could learn programming.
- **1970–1990.** In 1972, Dennis Ritchie developed the C language at Bell Laboratories. The UNIX operating system was largely rewritten in C. Because C was close to the hardware yet portable across processors, it became the basic language of systems programming. In 1983, Bjarne Stroustrup announced C++, which added the concepts of classes and objects to C.
- **After 1990.** Python appeared in 1991, Java and JavaScript in 1995, and C# in 2000. What these languages share are managed environments that take memory management away from the programmer, and suitability for the web. Rich libraries and open-source ecosystems also helped them spread. More recently, Go (2009) focused on concurrent server programs and Rust (version 1.0 in 2015) on safe systems programming.

One trend stands out in this history: each new generation took over a job the previous one left to the programmer. Assembly took binary encoding off the programmer's hands, FORTRAN took register management, and Java took memory management. What was left to the programmer moved a little closer to the problem itself.

4.3.3.3 Application areas and choosing a language

Programs run in very different environments, and each environment has different needs. The main application areas are these:

- **Web:** interfaces that run in the browser (the client), server programs that answer requests, and web services.
- **Mobile:** phone and tablet apps.
- **Data and artificial intelligence:** data analysis, statistical modeling, machine learning, and automation.
- **Systems:** operating systems, device drivers, and the embedded software inside a household appliance.
- **Scientific computing:** simulation and engineering solutions; the behavior of a bridge under load or the frequency response of a circuit.
- **Games:** game engines, graphics, and interaction with the user.

In each area some languages stand out (Figure 4.3.16). For interfaces that run in the browser, JavaScript (and its typed extension TypeScript) is almost the only choice, because it is the language browsers run directly. Android apps are written in Kotlin or Java and iOS apps in Swift; the official

tools of these platforms support these languages. In data analysis, Python and R are common thanks to their rich libraries. In embedded systems, memory and processing power are limited, so C, C++, and Rust are preferred. On enterprise servers, Java, C#, and Go are common.

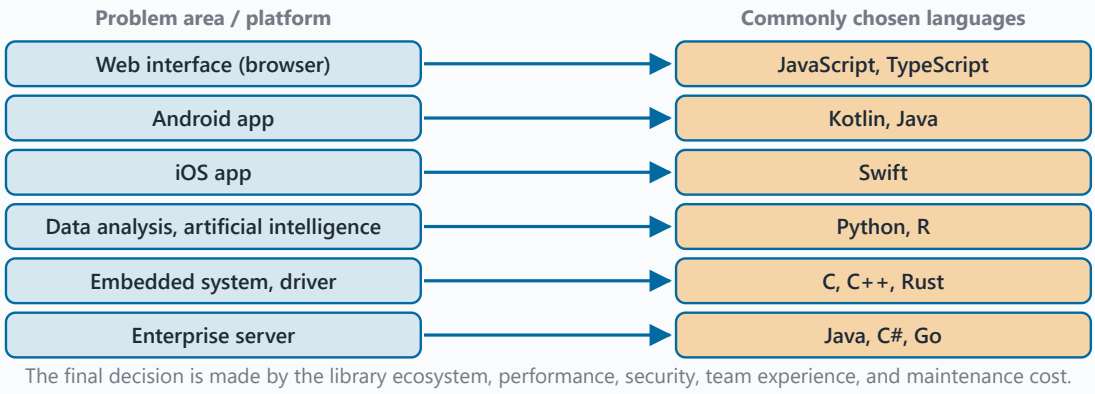


Figure 4.3.16: Languages often chosen by problem area and platform. The final decision is made by the library ecosystem, performance, security, team experience, and maintenance cost.

When choosing a language, first consider the problem area and the platform the program will run on. Then ask these questions. Are there ready-made libraries that do the needed work? Is the language's community large enough to find answers to questions? If the program will be maintained for years, is it easy to find developers who know the language? How important are speed and memory use? Does the language have a weakness from a security point of view? Which language does the team already know? How will the program be distributed?

Example 4.3.8

A group of electrical and electronics engineering students will do two projects: (a) a program that runs on a microcontroller, reads a temperature sensor, and switches a fan on and off, and (b) an analysis that plots a week of data collected from the sensor and finds its mean and maximum. Which languages are suitable?

1. (a) A microcontroller may have only a few kilobytes of memory and no operating system. The program must access hardware registers directly. These conditions call for a compiled language close to the low level: C or C++. Rust is also an increasingly common option.
2. (b) The program will run once on a laptop; speed is secondary. Ready-made libraries for graphics and statistics and quick experimentation matter. Python (or R) is suitable.
3. Choosing different languages for the two projects is not a contradiction. The choice follows from the needs of the problem and the platform.

Common mistake

Choosing a language only because it is popular. The most popular language may not fit on a microcontroller, or the library you need may not exist in that language. Popularity is a plus in terms of community and abundance of resources, but it alone is not enough for the right choice.

4.3.3.4 Kinds of languages

Languages are also grouped by purpose of use. A language can fall into more than one group; Python is both general-purpose and a scripting language.

Kind	Characteristic	Example languages	Example line
General-purpose	Offers variables, decisions, loops, and functions for any kind of problem	Java, C, Python	`for (int i = 0; i < 3; i++)`
Domain-specific	Focuses on a single job	SQL, Excel formulas	`SELECT name FROM student WHERE avg >= 60;`
Scripting	Automates repetitive tasks, usually interpreted	Python, JavaScript, Bash, PowerShell	`Copy-Item *.txt backup\`
Markup	Describes the structure of content; has no decisions or loops	HTML, XML, Markdown	`<h1>Title</h1>`

Table 4.3.6: Kinds of languages and examples

General-purpose languages offer the whole set of constructs such as variables, decisions, loops, and functions. With these constructs, in principle, any computation a computer can perform can be written. The `for` line in the table is the head of a loop that runs three times in Java.

Domain-specific languages do one job very well. SQL is designed for querying databases. The line in the table fetches the names of those in the student table whose average is 60 or above. The line says what is wanted, not how to find the result; the database management system decides how to scan the table. Formulas in Excel such as `=AVERAGE(B2:B30)` are also a domain-specific language: they compute in cells.

Scripting languages are used to automate repetitive tasks. The PowerShell command in the table copies all the `.txt` files in the current folder to the `backup` folder. Instead of copying hundreds of files by hand, this single line is run. Scripting languages are usually interpreted; a line you write can be tried at once.

Markup languages describe the structure of a document. The `<h1>Title</h1>` in HTML says "this text is a first-level heading." Markup languages have no decisions or loops; they cannot do different things depending on a condition and cannot repeat a task. For this reason they are not counted as programming languages. In a web page, HTML defines the structure, CSS the appearance, and JavaScript the behavior; of these three, only JavaScript is a programming language.

Block-based visual languages can be added to these groups. In languages such as Scratch, you do not type commands; you drag colored blocks and snap them together. Because blocks can only fit together in rule-abiding ways, syntax errors cannot be made. This property shifts attention from syntax to the algorithm when one is just beginning to learn programming.

Common mistake

Counting HTML as a programming language. HTML defines the heading, paragraph, and link on a page but computes nothing: it cannot add two numbers, test a condition, or repeat a task. The "programmed" part of a web page is the JavaScript code that works alongside it.

4.3.3.5 Programming paradigms

There are several ways of thinking about solving the same problem. One counts off the steps in order, one sees the problem as objects talking to each other, and one builds a chain of functions that transform data. These ways of thinking are called programming paradigms.

Definition 4.3.10

A **programming paradigm** is a fundamental approach to how a program is organized. **Imperative programming** specifies step by step how operations are to be done and changes the state of the program with those steps. **Object-oriented programming** combines data and the behavior that works on that data in objects. **Functional programming** builds the program as a composition of functions that work on immutable data.

Figure 4.3.17 compares the three approaches on the same task: finding the sum of the array grades = {70, 90, 80}. The Java program below contains all three approaches.

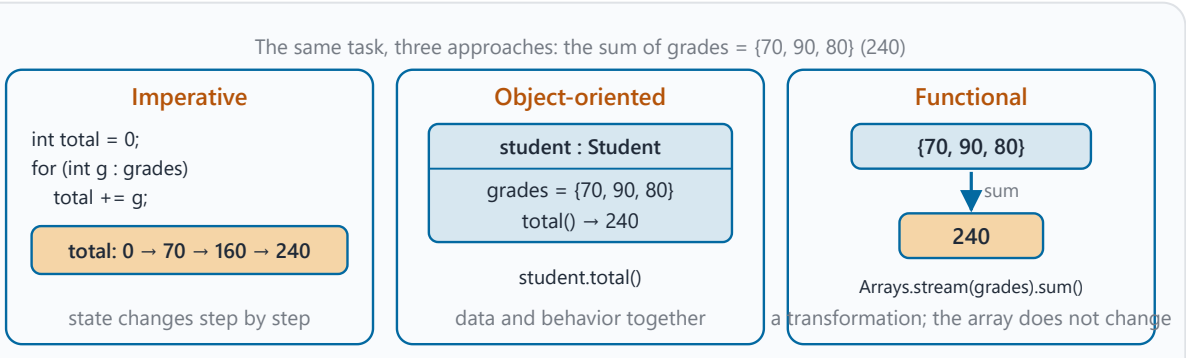


Figure 4.3.17: The same task, three approaches. The imperative loop changes state step by step; in the object-oriented solution the object's own method computes the sum; the functional solution sums the array with a transformation without changing it.

```
import java.util.Arrays;

public class Paradigms {
    static class Student {
        private final int[] grades;
        Student(int[] grades) { this.grades = grades; }
        int total() {
            int t = 0;
            for (int g : grades) t += g;
            return t;
        }
    }

    public static void main(String[] args) {
        int[] grades = {70, 90, 80};
        /* imperative */
        int total = 0;
        for (int g : grades)
            total += g;
        System.out.println(total);
        /* object-oriented */
        Student student = new Student(grades);
        System.out.println(student.total());
        /* functional */
        System.out.println(Arrays.stream(grades).sum());
        System.out.println(Arrays.toString(grades));
    }
}
```

```
240
240
240
[70, 90, 80]
```

The imperative solution starts a variable named total at 0 and changes it on every turn of the loop. The successive values of the total are 0, 70, 160, and 240. The program reads like a recipe: "first do this, then do that." C and assembly are fundamentally imperative languages.

In the object-oriented solution, the grades live inside a `Student` object. The job of computing the total belongs to the same object: `student.total()`. The code that uses the object does not need to know how the total is computed; it just asks the object. This arrangement pays off as programs grow. Everything about the student lives in one place, and when a change is needed it is clear where to look. Java, C++, and C# are object-oriented languages.

In the functional solution, the expression `Arrays.stream(grades).sum()` turns the array into a stream of values and sums the stream. No variable holds an intermediate state, and the array itself does not change. The last line of the program confirms this: the array is still `[70, 90, 80]`. The

functional approach favors functions that always give the same output for the same input and change nothing outside themselves. Such functions are easy to test and to combine. Haskell is a purely functional language; Python, JavaScript, and Java also offer functional features.

Many of today's languages support more than one paradigm. All three solutions can be written in Java, and in Python too:

```
grades = [70, 90, 80]
total = 0
for g in grades:
    total += g
print(total)
print(sum(grades))
from functools import reduce
print(reduce(lambda a, b: a + b, grades, 0))
```

```
240
240
240
```

The `reduce` function in Python applies a small function that adds two numbers to the elements of the array in turn, starting from 0: *((0 + 70) + 90) + 80 = 240*. Which approach to choose depends on the structure of the problem and the habits of the team. Large programs often use all three approaches together.

Example 4.3.9

The largest of the numbers in an array is to be found. How do the imperative and functional approaches express this job? Let the array be {12, 45, 7, 30}.

1. Imperative: a variable named `max` starts at the first element of the array (12). Each element is compared with `max`; if it is larger, `max` is updated. The state changes step by step: 12 → 45 → 45 → 45. The result is 45.
2. Functional: `Arrays.stream(array).max().getAsInt()` in Java, `max(array)` in Python. The expression says "the maximum of the array"; the order of comparisons and the intermediate variable are invisible to the programmer. The array does not change, and the result is again 45.
3. In both solutions the computer makes the same comparisons. The difference lies in how the program reads: one puts "how" in front, the other "what."

4.3.3.6 Static and dynamic typing

Every value has a type: 5 is an integer, `"five"` is a string, and `true` is a Boolean. Languages split in two according to when types are checked.

Definition 4.3.11

Static typing is the discipline in which a variable's type is known in the source code and checked at compile time: Java, C, C#. **Dynamic typing** is the discipline in which the type is carried by the value rather than the variable and is checked at run time: Python, JavaScript.

In Java, a variable's type is written when it is declared: `int x = 5;`. From then on x can hold only integers. The line `x = "five";` is caught in the compiler's semantic analysis phase: "incompatible types: String cannot be converted to int". The error is found before the program ever runs.

In Python, a variable's type is not declared. The same name can refer first to a number and then to a text. The type is carried by the value itself:

```
x = 5
print(type(x).__name__)
x = "five"
print(type(x).__name__)
print("a" + 1)
```

```
int
str
Traceback (most recent call last):
  File "dynamic.py", line 5, in <module>
    print("a" + 1)
          ~~~~^~~
TypeError: can only concatenate str (not "int") to str
```

The output of the first two lines says that x first holds a value of type `int` and then one of type `str` (string). Both are valid. The `"a" + 1` on line 5, however, tries to add a number to a string and raises a `TypeError`. What matters is when the error appears. When Python parses the file it sees no problem with this line, because the line's syntax is correct. The error appears only when the line runs, after the output of the two earlier `print` lines has been written to the screen. If this line were in a rarely executed branch of the program, the error might go unnoticed for months.

Some languages silently convert types instead of reporting an error on a type mismatch. JavaScript is the well-known example. The following lines were run with Node.js:

```
console.log("5" * 2);
console.log("5" + 2);
console.log(typeof ("5" * 2), typeof ("5" + 2));
console.log("5" - 2);
```

```
10
52
number string
3
```

In `"5" * 2`, because multiplication can be done only on numbers, JavaScript converts the string `"5"` to the number 5 and gets 10. In `"5" + 2`, however, since `+` also means string concatenation, 2 is converted to a string and the result is `"52"`. The third line shows the types of the two results: one is a number, the other a string. The same two values were converted to different types depending on the operation. Silent conversions of this kind are convenient in short scripts but lead to hard-to-find logic errors in large programs.

Characteristic	Static (Java)	Dynamic (Python)
Where does the type live?	In the variable, written in the declaration	In the value; there is no declaration
When is a type error found?	At compile time, before the program runs	When that line runs
Can the same name hold values of different types?	No	Yes
Strength	Errors are found early, and the IDE helps more	Short code, quick experimentation

Table 4.3.7: Comparison of static and dynamic typing

Common mistake

Thinking dynamic typing means "no types." In Python every value has a definite type, and Python enforces the type rules: `"a" + 1` raises an error. What dynamic typing lacks is not the type rules but the checking of those rules before the program runs.

4.3.3.7 From pseudocode to real languages

When an algorithm is first written in pseudocode and then translated into a language, every construct in the pseudocode has a counterpart in the language. Table 4.3.8 gives the Java and Python counterparts of the most commonly used constructs.

Pseudocode	Java	Python
`read x`	`x = in.nextInt();`	`x = int(input())`
`print "Sum: ", t`	`System.out.println("Sum: " + t);`	`print("Sum:", t)`
`x ← x + 1`	`x = x + 1;`	`x = x + 1`
`if x > 0 then … else …`	`if (x > 0) { … } else { … }`	`if x > 0: … else: …`
`for i ← 1 to n`	`for (int i = 1; i <= n; i++)`	`for i in range(1, n + 1):`
`repeat while the condition is true`	`while (condition) { … }`	`while condition:`

Table 4.3.8: Java and Python counterparts of pseudocode constructs

Pseudocode uses `←` for assignment, because the `=` sign means equality in mathematics. Java and Python write assignment with `=` and the equality comparison with `==`. The `in` in Java is a `Scanner` object that reads from the keyboard. In Python, `input()` always returns a string; if a number is wanted, it must be converted with `int(...)`.

The most visible difference between the two languages is how blocks are delimited. Java marks a block with curly braces (`{ }`) and ends each statement with `;`. Python wants neither; indentation determines the block. So in Python indentation is not just appearance but part of the program's meaning:

```
x = -3
if x > 0:
    print("positive")
print("done")
if x > 0:
    print("positive")
    print("done")
```

```
done
```

Because x = −3, both conditions are false. In the first `if` block, `print("done")` is not indented, so it is outside the block and runs regardless of the condition. In the second block the same line is indented, so it is inside the block and does not run because the condition is false. The only difference between the two parts is four spaces; only one "done" appears in the output.

Example 4.3.10

How is pseudocode that finds the sum of the numbers from 1 to n translated into Java and Python for n = 5?

```
sum ← 0
for i ← 1 to n
    sum ← sum + i
print "Sum: ", sum
```

The Java version (n = 5 is taken as a constant):

```
public class SumLoop {
    public static void main(String[] args) {
        int sum = 0;
        for (int i = 1; i <= 5; i++) {
            sum = sum + i;
        }
        System.out.println("Sum: " + sum);
    }
}
```

The Python version:

```
n = 5
sum = 0
for i in range(1, n + 1):
    sum = sum + i
print("Sum:", sum)
i = 1
while i <= 3:
    print(i)
    i = i + 1
print("done")
```

The Python file also prints 1, 2, 3 with a `while` loop. The outputs:

```
Sum: 15
```

```
Sum: 15
1
2
3
done
```

The sum changes on each turn as follows: *0 + 1 = 1*, *1 + 2 = 3*, *3 + 3 = 6*, *6 + 4 = 10*, *10 + 5 = 15*. The expression `range(1, n + 1)` in Python starts at 1 and stops before reaching n + 1; that is, it produces 1, 2, 3, 4, 5. Had `range(1, n)` been written, 5 would be skipped and the sum would be 10.

4.3.4 Problem solving

Writing a program does not begin at the keyboard. First you understand what the problem is, then you solve a few examples by hand, and only at the end do you write code. When this order is skipped, the result is often programs that "run but do the wrong job."

4.3.4.1 From problem to program

The example problem is a simple one: a program that computes the area of a rectangle. Figure 4.3.18 shows this problem turning into a program in five stages. The output of each stage is the input of the next.

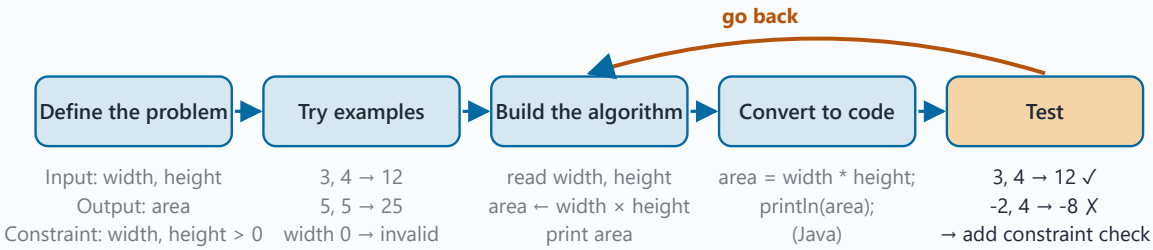


Figure 4.3.18: Five stages from problem to program. When a negative input gives a wrong result in testing (orange arrow), we go back to the algorithm and add a constraint check.

1. **Define the problem.** What is wanted becomes clear: the area of a rectangle. The program's **input** is the width and height, and its **output** is the area. There is also a **constraint**: a length cannot be zero or negative, so width and height must be greater than 0. Constraints are often not written down explicitly in the problem; whoever defines the problem has to find them.
2. **Try examples.** A few examples are solved by hand: *3 × 4 = 12*, *5 × 5 = 25*. A boundary case is also considered: what should happen if the width is 0? By the constraint, this input is invalid. The examples solved by hand also supply the expected results for later checking whether the program works correctly.
3. **Build the algorithm.** The language-independent steps are written in pseudocode: read the width, read the height, compute the area, print the area.
4. **Convert to code.** The algorithm is translated into the chosen language. In Java, the computation line becomes `area = width * height;` and the printing line `System.out.println(area);`.

5. **Test.** The program is run with the examples solved by hand. For 3 and 4 it gives 12: correct. For −2 and 4 it gives −8: since an area cannot be negative, this is wrong. The error is not in the code but in the algorithm; the algorithm never checks the constraint. We go back to the algorithm and add the step "if width and height are not greater than 0, report an error."

The return in the last stage is a natural part of the process. The stages form not a straight line but a loop: if testing finds something missing in the coding or the algorithm, we go back to the relevant stage. The corrected algorithm is this:

```
read width
read height
if width > 0 and height > 0 then
    area ← width × height
    print "Area: ", area
else
    print "Invalid size"
```

The Java version of this algorithm is below. In Java, `&&` is the "and" operator: the result is true if both conditions are true.

```
import java.util.Scanner;

public class Area {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int width = in.nextInt();
        int height = in.nextInt();
        if (width > 0 && height > 0) {
            int area = width * height;
            System.out.println("Area: " + area);
        } else {
            System.out.println("Invalid size");
        }
    }
}
```

The program was run with four inputs: (3, 4), (−2, 4), (0, 5), and (5, 2). The outputs are, in order:

```
Area: 12
```

```
Invalid size
```

```
Invalid size
```

```
Area: 10
```

Common mistake

Starting to write code without defining the problem. A badly defined problem turns into a program that works correctly but does the wrong job. The request "a program that computes a grade average" leaves several questions unanswered. Are the grades weighted? How many grades will be entered? What happens if an invalid grade (−5, 120) is entered? A program written without asking these questions has to be changed on its first real use.

4.3.4.2 Tracing the area algorithm

The most reliable way to be sure that an algorithm works correctly is to trace it line by line on paper. At each step the values of the variables are written into a table. This table is called a **trace table**. Variables can be thought of as boxes in memory: the line `read width` writes the number coming from the keyboard into the box named "width." Boxes that have not yet been given a value are shown with "—" in the table.

Table 4.3.9 traces the algorithm on the input width = 3, height = 4.

Step	Line	width	height	area	Explanation / output
1	`read width`	3	—	—	3 is read and written into the width box
2	`read height`	3	4	—	4 is read
3	`if width > 0 and height > 0`	3	4	—	3 > 0 and 4 > 0 → true; the "then" branch is entered
4	`area ← width × height`	3	4	12	*3 × 4 = 12*
5	`print "Area: ", area`	3	4	12	Output: `Area: 12` ; the "else" branch is skipped

Table 4.3.9: Tracing the area algorithm, run 1 (width = 3, height = 4)

Table 4.3.10 traces the same algorithm on the input width = −2, height = 4. This is the input that gave a wrong result in the testing stage.

Step	Line	width	height	area	Explanation / output
1	`read width`	−2	—	—	−2 is read
2	`read height`	−2	4	—	4 is read
3	`if width > 0 and height > 0`	−2	4	—	−2 > 0 is false; the whole condition joined by "and" is false → the "else" branch

Step	Line	width	height	area	Explanation / output
4	`print "Invalid size"`	−2	4	—	Output: `Invalid size`; the area was never computed

Table 4.3.10: Tracing the area algorithm, run 2 (width = −2, height = 4)

In the second trace the area box stays empty from start to finish. The program never entered the "then" branch. The error found in testing is thus fixed: for −2, a warning is now printed instead of −8.

Example 4.3.11

A perimeter computation is to be added to the area algorithm: *perimeter = 2 × (width + height)*. How does the algorithm change? What are the outputs for width = 6, height = 1.5?

1. Two lines are added to the "then" branch: `perimeter ← 2 × (width + height)` and `print "Perimeter: ", perimeter`. The constraint check stays the same; for an invalid size the perimeter is not computed either.
2. Condition: 6 > 0 and 1.5 > 0 → true.
3. area = *6 × 1.5 = 9*.
4. perimeter: first the parentheses, *6 + 1.5 = 7.5*; then *2 × 7.5 = 15*.
5. Output: "Area: 9" and "Perimeter: 15". Without the parentheses, the multiplication would be done first and the result would be *2 × 6 + 1.5 = 13.5*.

Because width and height can be fractional with this input, the Java version must use double instead of int.

4.3.5 Software development

A small program such as the rectangle area is one person's work for an hour. A bank's mobile application or a hospital's appointment system, on the other hand, is the work of dozens of people over months or even years. At this scale, organizing the work is at least as important as the code itself.

4.3.5.1 The software development process

The path from problem to program is divided, at large scale, into six stages (Figure 4.3.19). The stages can be explained with an "app that computes a student's grade average."

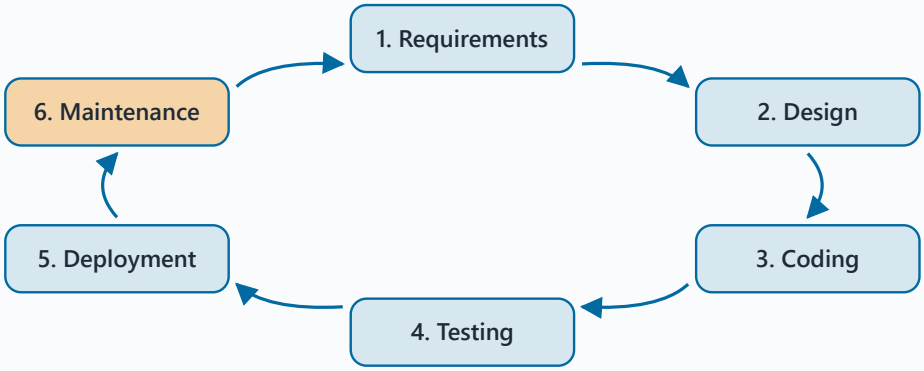


Figure 4.3.19: The software development process. Six stages from requirements to maintenance; a bug found in testing sends us back to coding, and new requests arriving in maintenance send us back to requirements.

1. **Requirements.** What does the user want? The student will enter the grade and credit of each course, and the app will show the weighted average. A grade must be between 0 and 100. The app will run on a phone. Requirements are written by talking with the user, and all later stages rest on this text.
2. **Design.** The screens, the data, and the algorithms are planned. What will be entered on which screen? Where will the grades be stored? By what formula will the average be computed? Flowcharts and pseudocode are the tools of this stage.
3. **Coding.** The design is written in a programming language.
4. **Testing.** Expected results are compared with actual results. For two courses, one with 60 points and 3 credits and one with 90 points and 2 credits, the expected average is $(60 \cdot 3 + 90 \cdot 2) / 5 = 360 / 5 = 72$. If the app shows 75 (the unweighted mean of the two grades), there is a bug and we go back to coding.
5. **Deployment.** The program is delivered to the user: it is published in an app store or an installation file is shared.
6. **Maintenance.** After use begins, new bugs are found and new requests arrive: "show the letter grade too." These requests are new requirements, and the cycle starts again.

Depending on the project, the stages are carried out sequentially or iteratively. Feedback from users can change the requirements and the design. Most of the effort spent over a software's lifetime goes not to the first version but to maintenance. That is why it matters to write code from day one so that it is readable and changeable.

Common mistake

Thinking the testing stage means "seeing that the program works." The aim of testing is not to prove that the program works correctly but to find the cases in which it works incorrectly. A good test suite includes normal inputs whose results have been computed by hand

beforehand, boundary cases (0, an empty list, the smallest and largest values), and invalid inputs.

4.3.5.2 Development approaches

There are different approaches to organizing the six stages (Figure 4.3.20).

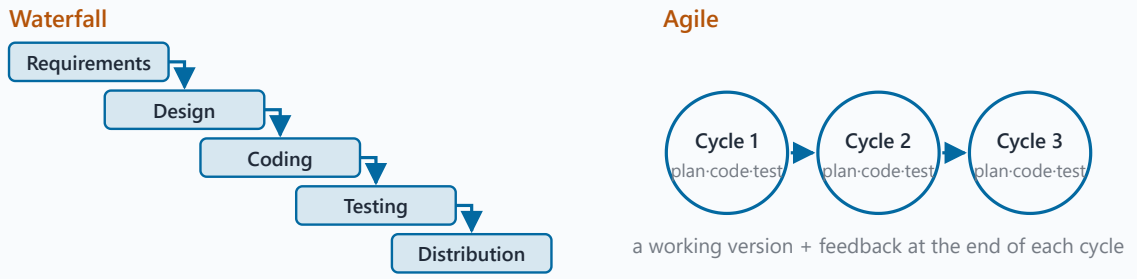


Figure 4.3.20: The waterfall model passes through the stages once, step by step. The agile approach proceeds in short cycles; at the end of each cycle there is a working version and user feedback.

In the **waterfall model** the stages are done in order and once each. Design does not start until the requirements are fully written, and coding does not start until the design is finished. A document is produced at the end of each stage. The strength of this arrangement is its clarity: who does what and when is known. Its weakness is that it is closed to change. If a gap in the requirements is noticed during coding, the cost of going back is high.

In the **agile** approach the work is divided into short cycles, each a few weeks long. In each cycle there is a little planning, a little coding, and testing. At the end of the cycle a working version is shown to the user and feedback is collected. A misunderstood requirement thus surfaces within a few weeks rather than months later. The price of this approach is the need for constant communication with the user.

In **prototyping**, a quick draft is prepared before starting on the real product. The draft has screens, but there may be no real computation or database behind them. If the user cannot fully explain what they want, the idea becomes clear once they see the draft. The purpose of the draft is to make the requirements visible. It must not be mistaken for the product itself and put straight into use.

Approach	Strength	Watch out for
Waterfall model	Stages and documents are clear	Going back to change things is costly
Agile	Gets feedback in short cycles	Requires constant communication
Prototyping	Makes vague requirements visible	A prototype must not be mistaken for the product

Table 4.3.11: Comparison of development approaches

Example 4.3.12

Which approach suits each of three projects? (a) Software that controls an aircraft's landing gear; the requirements have been fixed in advance by international standards. (b) A café's ordering app; the owner says "I'll decide when I see the screens." (c) A student club's events app; the features change every term at the members' request.

1. (a) The waterfall model. The requirements are fixed, and documenting every stage is mandatory for audit. Since no change is expected, the model's weakness is no problem here.
2. (b) Prototyping. The user can say what they want only when they see concrete screens. A quick draft clears up this vagueness.
3. (c) The agile approach. Requests keep changing; short cycles and the feedback collected at the end of each cycle suit this.

4.3.5.3 Development tools

A developer's daily work is a loop: write the code, translate and run it, test the result, find the bug if there is one, fix it, and try again. A different tool comes into play at each step of this loop (Figure 4.3.21).

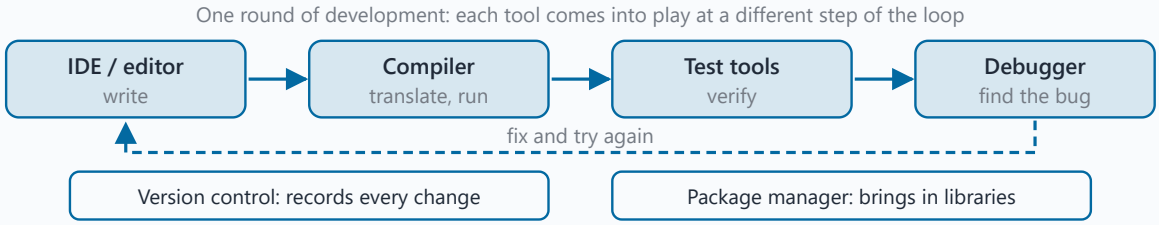


Figure 4.3.21: One round of development. Each tool comes into play at a different step of the loop; version control and the package manager support the whole loop.

- **Code editor and IDE.** Code can be written in a text editor, but programmers mostly use an **integrated development environment** (IDE). An IDE colors the code, completes names, underlines syntax errors as you type, and organizes the project's files. IntelliJ IDEA, Eclipse, and Visual Studio Code are common examples.
- **Compiler and runtime.** It translates and runs the program: javac and the JVM, the Python interpreter, a C compiler.
- **Debugger.** It stops the program at a chosen line, shows the current values of the variables, and steps through line by line.
- **Version control.** It records every change to the files and makes teamwork possible.

- **Testing tools.** They verify expected behavior automatically. The programmer writes once, "for this input this output is expected," and the tool reruns all the tests within seconds after every change. In Java, JUnit is the well-known tool for this.
- **Package manager.** It downloads the ready-made libraries (dependencies) the program uses, tracks their versions, and updates them: pip for Python, npm for JavaScript, Maven and Gradle for Java.

Common mistake

Assuming code the IDE does not underline is free of errors. An IDE can show syntax and type errors as you type, because it runs the first phases of the compiler in the background. It cannot see runtime and logic errors. The absence of a red underline says only that compilation will succeed.

4.3.5.4 Version control

If a student keeps copies of a file while writing an assignment as `hw_final.java`, `hw_final2.java`, `hw_REALLYFINAL.java`, they are trying to do version control by hand. This method gets muddled within days. Which copy has which change is forgotten; and if two people work on the same file, things become hopeless.

Definition 4.3.12

Version control is a system that stores the history of changes to files and organizes how several people work on the same files. A **commit** is the storing of the state of the files at a given moment together with who changed what and when. A **branch** is a line of changes that splits off from the main code and proceeds independently. A **merge** is the transfer of one branch's changes into another branch.

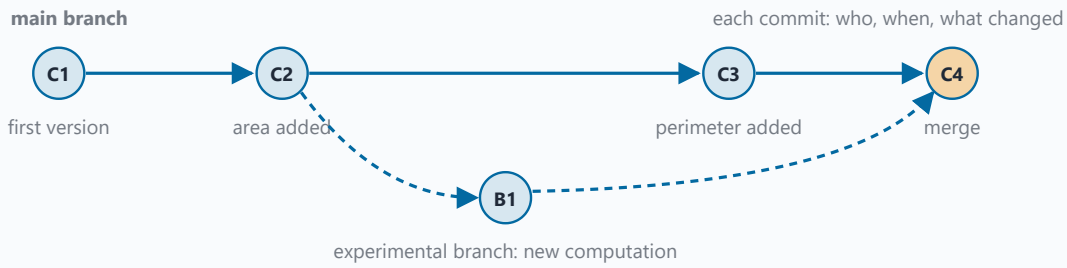


Figure 4.3.22: Four commits on the main branch. An experimental branch (B1) that splits off from the second commit tries a new computation and is merged into the main branch at the fourth commit.

Figure 4.3.22 shows the history of the rectangle program. C1 is the first version. In C2 the area computation was added, and in C3 the perimeter computation. After C2, a developer opened a branch (B1) to try a new computation and worked there without breaking the main code. When the experiment succeeded, the branch was merged into the main branch (C4). Had the experiment failed, the branch would have been deleted and the main branch never affected. If a change in C3 introduced a bug, the project can be returned to its C2 state with a single command.

The most common version control tool is Git. In Git the history above is created with a few commands. `git init` turns a folder into a repository. `git add` stages the changed files for the next commit. `git commit -m "area added"` creates the commit with its message. `git branch experiment` opens a new branch; `git merge experiment` merges that branch into the current one. `git log` lists all commits with their author, date, and message.

In teamwork, every developer has a copy of the repository on their own computer. Services such as GitHub and GitLab store the repository on the network and serve as a common meeting point between the copies. If two people changed different lines of the same file, Git merges the changes automatically. If they changed the same line, a **merge conflict** occurs. Git shows the two versions side by side and asks a person to choose which one stays.

Example 4.3.13

Ayşe and Mehmet are working from the same repository. Line 12 of the program is `double vat = price * 0.18;`. Ayşe changes this line to `price * 0.20` and commits. Mehmet changes the same line to `price * VAT_RATE` and adds the constant at the top of the file. What happens when they try to merge their changes?

1. The constant line added at the top of the file is only Mehmet's change; Git takes it without trouble.
2. Both changed line 12. Git cannot know which is correct and reports a conflict. Both versions of the line are shown, marked, in the file.
3. The two developers talk and decide the final form of the line: `price * VAT_RATE`, with the constant's value 0.20. The chosen form is committed and the merge is complete.

Common mistake

Thinking version control is a tool only for big teams. A student working alone also benefits from being able to go back in time. When a change breaks the program, returning to the last working version takes one command. The commit messages also help you remember later why the code was changed.

4.3.6 Basic programming constructs

However different languages may look, they are all built on the same few building blocks. Values are stored in named places and computed with expressions. Different paths are chosen according to conditions, and work is repeated. Frequently performed jobs are given names, and many values are gathered under a single name. The constructs are given first in pseudocode and then with their Java and Python counterparts.

4.3.6.1 Variables and data types

Definition 4.3.13

A **variable** is a named place in memory that holds a value; its value can change during the program. A **data type** determines how a value is represented in memory and which operations can be performed on it.

A variable can be thought of as a box with a name written on it (Figure 4.3.23). The `age` box holds 19, the `height` box holds 1.75, the `name` box holds "Ali," and the `passed` box holds the value true. When a new value is written into a box, the old one is erased. The box's type limits what can be put in it: text cannot be put in an integer box.

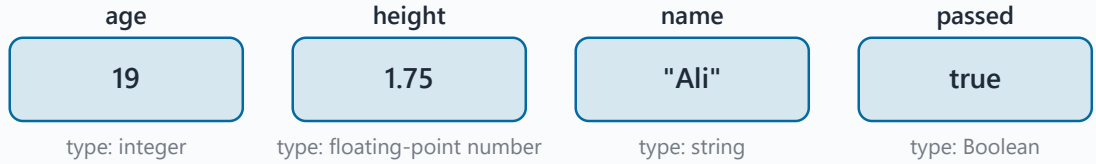


Figure 4.3.23: Named boxes in memory. Every box has a name, a value, and a type.

Type	What it holds	Example value	Java	Operations
Integer	A number without a fraction	19, −3, 0	`int`	Four arithmetic operations, remainder (mod)
Floating-point number	A number with a fraction	1.75, 81.67	`double`	Four arithmetic operations
String	A sequence of characters, text	"Ali"	`String`	Concatenation, length, getting a character
Boolean	True or false	true, false	`boolean`	and, or, not

Table 4.3.12: Basic data types and their Java counterparts

In code the decimal separator is the period: Java and Python do not accept the notation `1,75` as a number. In this note decimal fractions are written with a period everywhere (1.75).

The choice of data type follows from the meaning of the value. A student's name is a string because it consists of letters. A grade average is a floating-point number because it can be fractional, like 81.67. The answer to "did the student pass the course?" is Boolean because it can only be yes or no. The number of students is an integer because it cannot be fractional.

Giving variables meaningful names makes the purpose of the program visible. The line `a = b * c` says nothing; the line `area = width * height` explains itself. Names are the cheapest and most effective tool for readability.

Common mistake

Keeping a value that may be fractional in an integer type. In Java, the line `int avg = 245 / 3;` gives no error; it stores 81, and the fractional part is silently lost. For values such as averages, ratios, and measurements, `double` is used. At least one operand of the division must also be fractional: `245 / 3.0`.

4.3.6.2 Operator precedence

When an expression contains several operations, **operator precedence** determines which is done first. The rules in programming languages are the same as the mathematics rules learned at school:

1. What is inside parentheses is computed first.
2. Multiplication, division, and mod (remainder) are done before addition and subtraction.
3. Addition and subtraction are done last.
4. Operations at the same level are done from left to right.

The mod operation (`%` in Java and Python) gives the remainder of a division: 7 mod 3 is 1, the remainder of dividing 7 by 3. Table 4.3.13 shows these rules applied to five expressions.

Expression	Order of computation	Result
2 + 3 × 4	First *3 × 4 = 12*, then *2 + 12*	14
(2 + 3) × 4	First the parentheses *2 + 3 = 5*, then *5 × 4*	20
20 / 4 × 2	Same level, left to right: *20 / 4 = 5*, then *5 × 2*	10
10 − 4 − 3	Left to right: *10 − 4 = 6*, then *6 − 3*	3
7 mod 3 + 1	First *7 mod 3 = 1*, then *1 + 1*	2

Table 4.3.13: Examples of operator precedence

The compiler builds operator precedence into the expression's tree. Figure 4.3.24 shows the tree of the expression *2 + 3 × 4*: the multiplication is deeper, so it is done first; the addition is at the root, so it is done last.

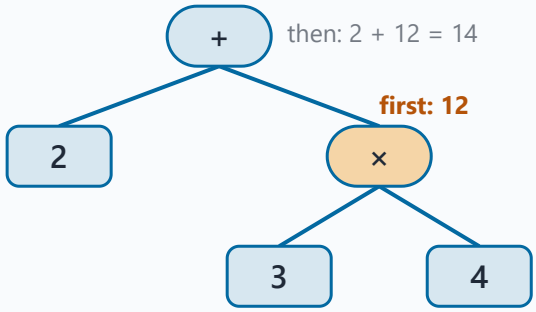


Figure 4.3.24: The tree of the expression *2 + 3 × 4*. The multiplication is deeper, so it is done first (12), then the addition at the root (14).

The expressions in the table give the same results in Java. The following program prints the table's five expressions and two extra expressions that test the left-to-right rule:

```
public class Precedence {
    public static void main(String[] args) {
        System.out.println(2 + 3 * 4);
        System.out.println((2 + 3) * 4);
        System.out.println(20 / 4 * 2);
        System.out.println(10 - 4 - 3);
        System.out.println(7 % 3 + 1);
        System.out.println(20 / (4 * 2));
        System.out.println(10 - (4 - 3));
    }
}
```

```
14
20
10
3
2
2
9
```

The last two lines show why the left-to-right rule matters. The expression *20 / 4 × 2* gives 10; written with parentheses as *20 / (4 × 2)*, the result is 2. In Java this second result should really have been 2.5: because the quotient of two integers is an integer, the fractional part of *20 / 8* is discarded. Similarly, *10 − 4 − 3 = 3* while *10 − (4 − 3) = 9*.

In Python the same expressions give the same results with one difference:

```
185.0
81.67
52.33
14 20 10.0 3 2
10
```

In the fourth line of the output, the expression `20 / 4 * 2` gives `10.0`, because in Python the `/` operator always produces a floating-point result. For integer division there is the `//` operator, which gives 10 as in the last line. The 185.0 on the first line is the result of the missing parenthesis. This error appears in Python exactly as it does in Java.

Example 4.3.14

A student's midterm grade is 60 and final grade is 80; the midterm weighs 40% and the final 60%. The programmer writes `avg ← midterm × 0.4 + final × 0.6`. Is the result correct? What would have come out if the same computation had been written as `avg ← midterm + final × 0.6 / 2`?

1. In the first expression the two multiplications are done first: *60 × 0.4 = 24* and *80 × 0.6 = 48*. Then the addition: *24 + 48 = 72*. No parentheses are needed, because multiplication is already done first. The result is correct.
2. In the second expression, multiplication and division come first, left to right: *80 × 0.6 = 48*, *48 / 2 = 24*. Then the addition: *60 + 24 = 84*. The program gives no error but prints 84; this is a logic error.

Common mistake

Forgetting the parentheses in the average formula: `avg ← a + b / 2`. This expression first divides b by two, then adds a. For a = 70, b = 90 the result is *70 + 45 = 115*; the correct value is *(70 + 90) / 2 = 80*. The program gives no error. For this reason, such errors are found only by testing. Using parentheses wherever you are unsure of the precedence both prevents the error and shows the reader your intent clearly.

4.3.6.3 Decisions and loops

If a program ran only from top to bottom in order, it would take the same steps for every input. Real problems, however, require behaving differently depending on conditions and doing the same work over and over. The two constructs that provide this are the decision and the loop (Figure 4.3.25).

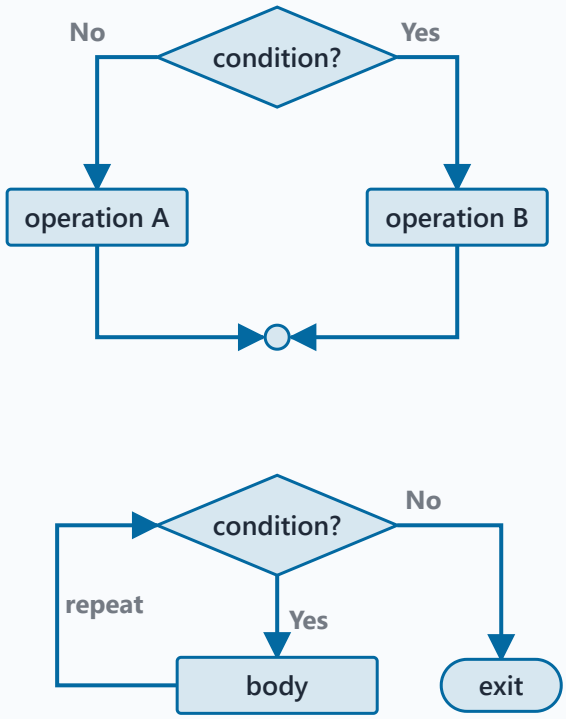


Figure 4.3.25: On the left, the decision: of the two paths, only one is followed depending on the condition. On the right, the loop: the body repeats while the condition is true; the arrow that goes back marks the loop.

Definition 4.3.14

A **selection**, or decision, chooses one of several paths of operations depending on whether a condition is true or false. A **loop** repeats a group of operations as long as a condition is true. The repeated group of operations is the loop's **body**, and each run of the body is an **iteration**.

In a decision, only one of the two paths is followed on each run. In the area algorithm, if the condition is true the area is computed, and if it is false a warning is printed; the two never happen together in the same run. In a loop, however, the flow returns to the condition at the end of the body. The condition is checked again at the start of every iteration. When the condition becomes false, the loop is exited and execution moves to the line after the loop.

For a loop to be able to end, the body must contain a change that affects the condition. If the condition is `i ≤ 3`, the body must push i above 3 someday. If there is no such change, the condition stays true forever and the loop never ends.

4.3.6.4 Tracing a loop: the infinite loop

The following pseudocode was written to print the numbers from 1 to 3. In the first version the programmer forgot line 4:

```
i ← 1
while i ≤ 3
    print i
    i ← i + 1      // missing in the first version
print "done"
```

Definition 4.3.15

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

Table 4.3.14 traces the first few steps of the first version. With i ← 1 the loop variable is set up. The condition *1 ≤ 3* is true, the body is entered, and 1 is printed. Since there is no other line in the body, we return to the condition. i is still 1, and the condition is true again. The state is exactly the same as in the previous iteration.

Step	Line	i	Condition	Body executions	Output
1	`i ← 1`	1	—	0	
2	`while i ≤ 3`	1	1 ≤ 3 → true	0	
3	`print i`	1	—	1	1
4	`while i ≤ 3`	1	1 ≤ 3 → true	1	
5	`print i`	1	—	2	1
…	…	1	always true	grows without bound	1, 1, 1, …

Table 4.3.14: Tracing the first version (line 4 missing)

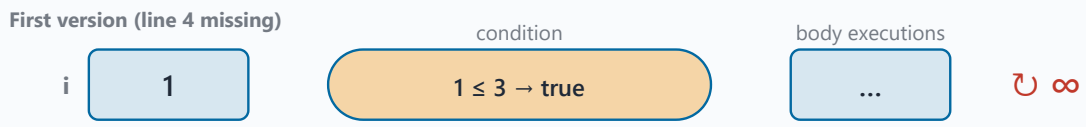


Figure 4.3.26: The infinite loop. i is always 1, the condition is always true, and the body runs an unbounded number of times.

Because nothing makes the condition false, the loop never ends (Figure 4.3.26). The screen fills with 1s. A program running in a console can be stopped only with Ctrl+C.

In the corrected version the line `i ← i + 1` is added to the body. Table 4.3.15 traces this version from start to finish.

Step	Condition check	Body	i (after the iteration)	Body executions
Startup	—	`i ← 1`	1	0
Iteration 1	1 ≤ 3 → true	1 is printed, i ← 2	2	1
Iteration 2	2 ≤ 3 → true	2 is printed, i ← 3	3	2
Iteration 3	3 ≤ 3 → true (equality)	3 is printed, i ← 4	4	3
Exit	4 ≤ 3 → false	The loop is exited; "done" is printed	4	3

Table 4.3.15: Tracing the corrected version

The loop variable now advances on every iteration. On the third iteration the condition $3 \le 3$ is still true, because ≤ includes equality. If this detail is overlooked, one might think the numbers will be printed only up to 1 and 2. When the loop finishes, the body has run 3 times and the condition has been checked 4 times: three true and one false. In a loop the condition is always checked once more than the body runs. The last check is the one that ends the loop. The Java version and its output:

```
public class CountLoop {
    public static void main(String[] args) {
        int i = 1;
        while (i <= 3) {
            System.out.println(i);
            i = i + 1;
        }
        System.out.println("done");
    }
}
```

```
1
2
3
done
```

Example 4.3.15

How many iterations does the following loop run, and what does it print? `k ← 1 ; while k < 20: print k; k ← k × 3` . What would happen if the line `k ← k × 3` in the body were forgotten?

1. Iteration 1: $1 < 20$ is true; 1 is printed, k = 3.
2. Iteration 2: $3 < 20$ is true; 3 is printed, k = 9.
3. Iteration 3: $9 < 20$ is true; 9 is printed, k = 27.
4. $27 < 20$ is false; the loop ends. It runs three iterations and prints 1, 3, 9; the condition is checked 4 times.

5. If the multiplication line in the body were forgotten, k would stay 1 forever. Since the condition would always be true, the loop would print 1 forever.

Common mistake

Changing the loop variable in the wrong direction. In the loop `i ← 1; while i ≤ 3: print i; i ← i − 1`, the body changes i, but not in the direction that makes the condition false. i decreases as 1, 0, −1, −2 and always stays below 3. It is not enough for the body to change something; the change must someday make the condition false.

4.3.6.5 Functions and arrays

As programs grow, two needs arise. The first is that the same job is done in many places in the program; the area of a rectangle may be computed in ten places. The second is storing many similar values; defining 40 separate variables for the grades of the 40 students in a class makes no sense. The function meets the first need and the array the second (Figure 4.3.27).

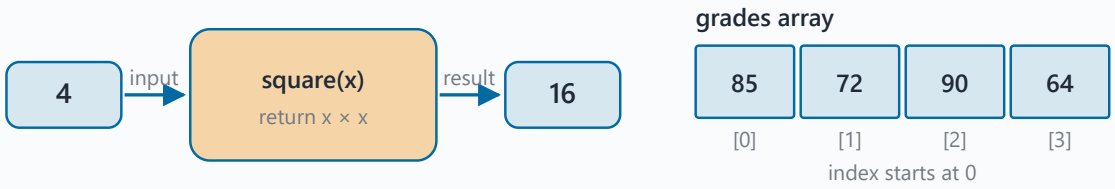


Figure 4.3.27: On the left, the square function: it takes the input 4 and returns *4 × 4 = 16*. On the right, the grades array: four cells, with indices starting at 0.

Definition 4.3.16

A **function** is a block of code that performs a specific task and is called by a name. It can take **parameters** as input and can **return** a result. The values given when calling the function are called **arguments**. An **array** is a structure that stores many values of the same type under a single name, by sequence number. The sequence number of an element is called its **index**.

A function names a task. The function `square(x)` is the name of the task "take the square of a number." The code that uses the function does not care how the square is computed; it just writes `square(4)` and gets 16. The formula lives in one place. If there is a bug in the formula, it is fixed once, and the fix is reflected everywhere the function is called.

In an array, values are selected by sequence number. In Java, C, and Python the sequence number starts at 0. In the four-element array `grades`, the first element is `grades[0]` = 85 and the last is `grades[3]` = 64. The last index is always length − 1. This rule may seem odd at first, but there is logic

to it: the index says how many steps the element is from the start of the array; the first element is zero steps from the start.

These two constructs divide a program into small pieces that can be tested separately. A program that computes a grade average can be split into three functions that read the grades, compute the average, and find the letter grade; the grades are also kept in a single array.

4.3.6.6 Tracing a function call

The following pseudocode defines the area function and calls it twice:

```
function area(width, height)
    result ← width × height
    return result
end
a ← area(3, 4)
b ← area(2, 5)
print a + b
```

Function calls are tracked in memory with a structure called the **call stack**. For every call, a new **stack frame** is added on top of the stack. The frame holds the function's parameters and its own variables. When the function finishes, its frame is removed from the stack. The frame at the top of the stack belongs to the function that is running at that moment. Table 4.3.16 traces the program step by step.

Step	Event	Main program	area frame
1	The definition (lines 1–4) is recorded, not run; the program starts at line 5	a = —, b = —	none
2	`area(3, 4)` is called; new frame, 3 → width, 4 → height	a = —, b = —	width = 3, height = 4
3	Body: *result ← 3 × 4 = 12*	a = —, b = —	width = 3, height = 4, result = 12
4	`return result` : 12 is sent back to the caller	a = —, b = —	about to be removed
5	The frame is removed; the returned value is assigned to a	a = 12, b = —	none
6	`area(2, 5)` is called; a brand-new frame	a = 12, b = —	width = 2, height = 5
7	Body: *result ← 2 × 5 = 10*	a = 12, b = —	width = 2, height = 5, result = 10
8	10 is returned, the frame closes	a = 12, b = —	about to be removed
9	The returned value is assigned to b	a = 12, b = 10	none
10	`print a + b` → 22	a = 12, b = 10	none

Table 4.3.16: Tracing a function call

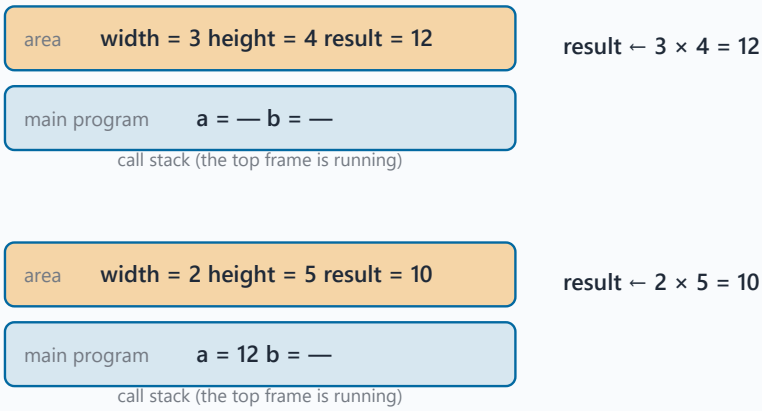


Figure 4.3.28: Two moments of the call stack. At the top the first call is running: the frame has width = 3, height = 4, result = 12. At the bottom the second call: a new frame for the same function, width = 2, height = 5, result = 10.

Three conclusions come out of the trace. First, a function definition by itself does nothing; it runs only when called. Second, every call opens its own frame. The first call's value result = 12 is not visible in the second call's frame, because the first frame has long been removed. The variables inside a function live only as long as that call lasts. Third, the main program does not see inside the function; it only receives the returned value. A function written once was used twice, and the area formula stays in a single place (Figure 4.3.28).

In Java, functions are called methods. The header `static int area(int width, int height)` says that the method takes two integers and returns an integer:

```
public class AreaFunction {
    static int area(int width, int height) {
        int result = width * height;
        return result;
    }

    public static void main(String[] args) {
        int a = area(3, 4);
        int b = area(2, 5);
        System.out.println(a + b);
    }
}
```

```
22
```

Example 4.3.16

`function perimeter(width, height): return 2 × (width + height)` is defined. The main program runs the line `c ← perimeter(3, 4) + perimeter(1, 1)`. How many frames are opened, and what is c?

1. First `perimeter(3, 4)` is called: a frame opens, width = 3, height = 4. *2 × (3 + 4) = 14* is returned and the frame closes.
2. Then `perimeter(1, 1)` is called: a new frame, width = 1, height = 1. *2 × (1 + 1) = 4* is returned and the frame closes.
3. The two returned values are added: *c = 14 + 4 = 18*. In total two frames were opened; the two are never on the stack at the same time.

Common mistake

Trying to use a variable defined inside a function outside the function. The variable `result` in the area function exists only in the function's frame; writing `print result` in the main program gives a "cannot find symbol" compile error in Java. A value the function produces gets out only through `return`.

4.3.6.7 Averaging with an array

The following pseudocode computes the average of four grades with an array and a loop:

```
grades ← [85, 72, 90, 64]
total ← 0
for i ← 0, 1, 2, 3
    total ← total + grades[i]
avg ← total / 4
print "Average: ", avg
```

The variable that holds the sum is called an **accumulator**. The accumulator is initialized to 0 before the loop, and one more value is added on every iteration. If no initial value were given, the first value of the total would be undefined. Table 4.3.17 traces each iteration of the loop.

Step	i	grades[i]	Operation	total
Startup	—	—	`total ← 0`	0
Iteration 1	0	85	*0 + 85*	85
Iteration 2	1	72	*85 + 72*	157
Iteration 3	2	90	*157 + 90*	247
Iteration 4	3	64	*247 + 64*	311

Step	i	grades[i]	Operation	total
After the loop	—	—	*avg ← 311 / 4*	311 (avg = 77.75)

Table 4.3.17: Tracing the average of the grades (grades = [85, 72, 90, 64])

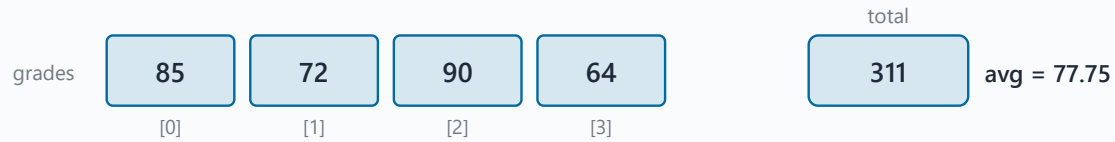


Figure 4.3.29: The array and the accumulator after the loop finishes: total 311, average 77.75.

The loop ends after four iterations. The last index is 3, that is, length − 1. The result is printed as "Average: 77.75" (Figure 4.3.29). In the Java program the loop bound is written not as a fixed 4 but as the array's length, `grades.length`. That way, when a grade is added to the array, the loop adapts by itself. The division is also done with `4.0` so that the result is not rounded to an integer:

```
public class Average {
    public static void main(String[] args) {
        int[] grades = {85, 72, 90, 64};
        int total = 0;
        for (int i = 0; i < grades.length; i++) {
            total = total + grades[i];
        }
        double avg = total / 4.0;
        System.out.println("Average: " + avg);
    }
}
```

```
Average: 77.75
```

In Python the same computation is shorter. The built-in functions `sum` and `len` hide the loop from the programmer:

```
grades = [85, 72, 90, 64]
total = 0
for i in range(4):
    total = total + grades[i]
avg = total / 4
print("Average:", avg)
print(sum(grades) / len(grades))
```

```
Average: 77.75
77.75
```

What would happen if the loop ran one iteration too many? An array with four elements has no cell number 4. Java reports this with a runtime error:

```
public class OutOfBounds {
    public static void main(String[] args) {
        int[] grades = {85, 72, 90, 64};
        System.out.println(grades[3]);
        System.out.println(grades[4]);
    }
}
```

```
64
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds
for length 4
        at OutOfBounds.main(OutOfBounds.java:5)
```

`grades[3]` is read without trouble and 64 is printed. When `grades[4]` is requested, the JVM stops the program. The phrase "Index 4 out of bounds for length 4" in the message means "in an array of length 4, index 4 is outside the bounds." Writing the loop bound as `i <= grades.length` is the most common cause of this error; the correct form is `i < grades.length`.

Example 4.3.17

In the array temps = [18, 21, 25, 19, 22], the days above 20 are to be counted. How is the trace done with a counter and a loop? How many days are found?

1. `count ← 0` . i = 0: temps[0] = 18; *18 > 20* is false, count stays 0.
2. i = 1: 21 > 20 is true, count 1.
3. i = 2: 25 > 20 is true, count 2.
4. i = 3: 19 > 20 is false, count stays 2.
5. i = 4: 22 > 20 is true, count 3. The last index is *5 − 1 = 4*; the loop ends. The result is 3 days.

In this example there is a decision inside the loop. The loop visits every element, and the decision chooses which element gets counted.

4.3.6.8 Libraries and APIs

Taking a square root, sorting a list, or reading a file are jobs every programmer often needs. If every programmer had to write them from scratch, it would waste time and be a source of bugs. For this reason languages come with ready-made, tested functions.

Definition 4.3.17

A **library** is a collection of ready-made functions that others have written and tested. An **API** (application programming interface) defines the functions a library or a service exposes to the outside, along with their inputs and return values.

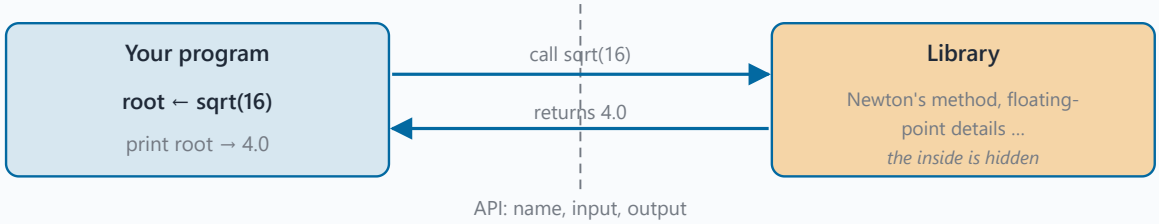


Figure 4.3.30: The program calls a library function through the API: the call sqrt(16) returns 4.0. The inside of the library is hidden.

An API is like a contract (Figure 4.3.30). The API of the square root function says three things: the function's name, that it takes a number, and that it returns the square root of that number. Which method is used inside the function and how floating-point details are handled do not concern the user. Even if the library's authors replace the inside with a faster method, programs that use it keep working unchanged as long as the API stays the same.

Call	Language	Result
`Math.sqrt(16)`	Java	4.0
`Math.max(3, 8)`	Java	8
`len("Giresun")`	Python	7

Table 4.3.18: Examples of library calls

The results in the table were taken from real runs. In Java, `Math.sqrt` always returns a `double`; that is why 4.0 is printed, not 4. In Java the length of a string is found not with a function but with the string's own method: `"Giresun".length()`.

```java
public class Libraries {
    public static void main(String[] args) {
        System.out.println(Math.sqrt(16));
        System.out.println(Math.max(3, 8));
        System.out.println("Giresun".length());
    }
}
```

```
4.0
8
7
```

```python
import math
print(math.sqrt(16))
print(max(3, 8))
print(len("Giresun"))
```

```
4.0
8
7
```

The concept of an API is not limited to libraries. A weather app calls the API that a meteorological service offers over the network: it sends the city name and gets the temperature. The people who wrote the app do not know what goes on in that service's servers; they only read the API's documentation.

Common mistake

Using a library function without knowing its return type. In Java, `Math.sqrt(16)` returns 4.0; assigning the result to an integer with `int k = Math.sqrt(16);` gives a compile error, because a double value is not converted to int automatically. The API documentation states what each function takes and returns; look before using it.

4.3.7 Debugging

Every programmer makes mistakes; what sets an experienced programmer apart is knowing an orderly way to find the error. Finding and removing an error in a program is called **debugging**.

4.3.7.1 Debugging techniques

Debugging is not a guessing game but an investigation that proceeds step by step. The following techniques are the tools of this investigation.

1. **Reproduce the bug.** Does the same input trigger the error every time? Starting the search before finding an input that brings out the error on demand is wasted effort. This input is also needed to check that the error is really gone after the fix.
2. **Print intermediate values.** Before and after the suspicious line, the values of the variables are printed to the screen. The first place where the expected and the actual value part ways is the location of the bug.
3. **Set a breakpoint.** In the debugger, a **breakpoint** is placed on a line. The program stops when it reaches that line, and all the variables at that moment can be inspected.
4. **Step through.** After the program stops, it is advanced line by line. After each line, look at which value has changed.
5. **Narrow the search.** Half of the code is temporarily disabled and you find which half contains the bug. Then that half is split in two again. In a hundred lines of code, the bug comes down to a single line in seven splits, because $2^7 = 128 \geq 100$.
6. **Explain it aloud.** Explaining the code line by line to someone, even to a toy on the desk, often reveals the bug. While explaining you say "this line does this," and suddenly you notice that the line actually does something else.

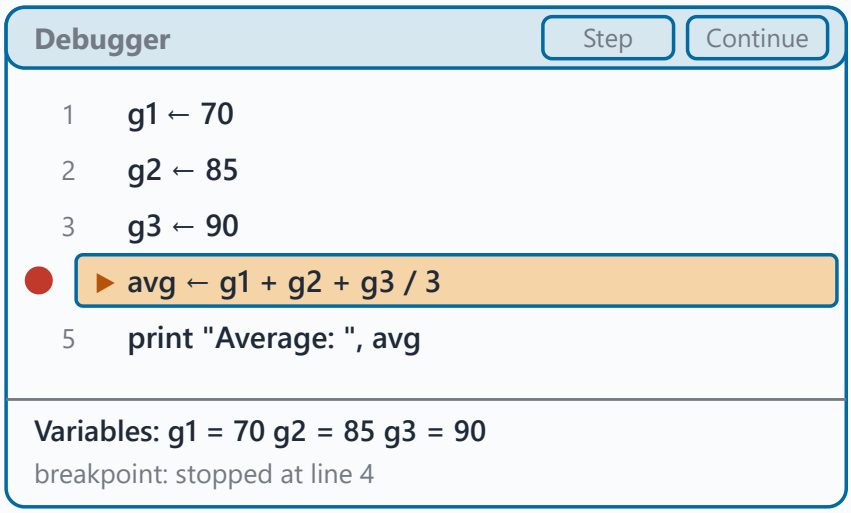


Figure 4.3.31: A debugger window. The program has stopped at the breakpoint on line 4; the variables panel shows g1 = 70, g2 = 85, g3 = 90.

Figure 4.3.31 imitates the look of a debugger. The program has stopped at the breakpoint on line 4 and has not yet run that line. The variables panel shows the three grades. The line is `avg ← g1 + g2 + g3 / 3`. When the "Step" button is pressed, first *g3 / 3 = 30* is computed, then *70 + 85 + 30 = 185*.

The average comes out as 185; the expected value is *245 / 3 ≈ 81.67*. The debugger shows at a glance that the problem lies in the order of the division and the addition.

> **Common mistake**
>
> Hiding the symptom of a bug without finding its cause. Because the average comes out as 185, dividing the result by 2.27 gives about 81.5 for this input but produces a wrong result for every other input. The fix is made at the root cause of the bug: the missing parenthesis is added.

4.3.7.2 Finding bugs by tracing

The following pseudocode was written to find the largest of three numbers. The variable max holds the "largest" value:

```
read a, b, c
max ← 0
if a > max then max ← a
if b > max then max ← b
if c > max then max ← c
print "Largest: ", max
```

For the first test, 7, 12, and 5 are chosen; the expected output is 12. Table 4.3.19 traces the program.

Line	Condition	max
`max ← 0`	—	0
`if a > max`	7 > 0 → true	7
`if b > max`	12 > 7 → true	12
`if c > max`	5 > 12 → false	12
`print`	—	Output: `Largest: 12` ✓

Table 4.3.19: Test 1 (a = 7, b = 12, c = 5; expected 12)

The output is the same as expected. But a single successful test does not prove the program correct. The second test tries a boundary case: let all the numbers be negative, −5, −2, and −9. The expected output is −2 (Table 4.3.20).

Line	Condition	max
`max ← 0`	—	0
`if a > max`	−5 > 0 → false	0
`if b > max`	−2 > 0 → false	0
`if c > max`	−9 > 0 → false	0

Line	Condition	max
`print`	—	Output: `Largest: 0` ✗

Table 4.3.20: Test 2 (a = −5, b = −2, c = −9; expected −2)

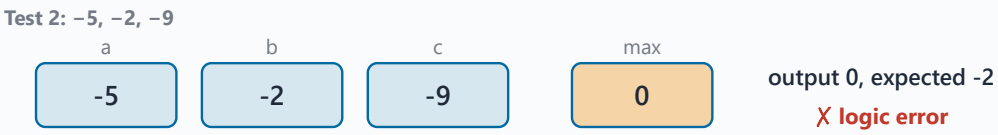


Figure 4.3.32: The end of Test 2. max never changed; the output is 0 but −2 was expected: a logic error.

All three comparisons are false, because no input is greater than 0; max never changes and the output is 0 (Figure 4.3.32). None of the inputs is 0. The trace points to the culprit: the initial value on line 2 comes from outside the inputs. 0 is not an initial value that is "no larger than any number"; it works only for positive numbers.

The fix is to take the initial value from one of the inputs: `max ← a`. Because max now starts with an input, the result is always one of the inputs. There is also no need to compare a with itself. The corrected program is traced again with Test 2: max starts at −5; *−2 > −5* is true, max = −2; *−9 > −2* is false. The output is −2 ✓. After a fix, the earlier tests must be repeated too, because the fix may have broken a case that worked before. Test 1 is tried again and still gives 12.

The buggy and corrected Java versions of the same program were run with the same inputs:

```
import java.util.Scanner;

public class LargestBuggy {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int a = in.nextInt(), b = in.nextInt(), c = in.nextInt();
        int max = 0;
        if (a > max) max = a;
        if (b > max) max = b;
        if (c > max) max = c;
        System.out.println("Largest: " + max);
    }
}
```

Outputs for the inputs 7 12 5 and −5 −2 −9:

```
Largest: 12
```

```
Largest: 0
```

```
import java.util.Scanner;

public class Largest {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int a = in.nextInt(), b = in.nextInt(), c = in.nextInt();
        int max = a;
        if (b > max) max = b;
        if (c > max) max = c;
        System.out.println("Largest: " + max);
    }
}
```

The corrected version for the same two inputs and an input with equal values (3, 3, 1):

```
Largest: 12
```

```
Largest: -2
```

```
Largest: 3
```

Example 4.3.18

Which test inputs should be chosen for the corrected largest-number program? What does each test?

1. Inputs in which the largest number is in each of the three positions: (12, 7, 5), (7, 12, 5), (7, 5, 12). A typo in one comparison line shows up only when the largest value is in that position.
2. All negative: (−5, −2, −9). It catches the initial value error.
3. Equal values: (3, 3, 1) and (4, 4, 4). Writing `>=` instead of `>` would not change the result, but the case of equality is at least exercised.
4. An input containing zero: (0, −1, −3). The expected result is 0; the buggy version also gives 0. This test alone cannot catch the buggy version; it is a case in which the wrong initial value happens to give the right result.

4.3.7.3 Common mistakes

Some mistakes are made so often that they have names. Table 4.3.21 lists them with examples and consequences.

Mistake	Example	What happens?
Missing or wrong initial value	`total ← total + x` without initializing total	Java refuses to compile; in some languages the computation uses a random value
Off-by-one error (one too few / one too many)	`for i ← 1 to n − 1`	The last element is skipped, or one extra iteration runs
Infinite loop	The loop variable is not updated in the body	The program never ends
Division by zero	`avg ← total / n` , n = 0	Runtime error
Going outside the array	`grades[4]` in an array of 4 elements	Runtime error
Forgetting parentheses	`avg ← a + b / 2`	Wrong result (logic error)
Not validating input	Accepting −3 as an age	A meaningless result, errors in later steps

Table 4.3.21: Common mistakes

Languages behave differently on the initial-value mistake. Java does not allow a local variable that has not been given a value to be read:

```
public class Uninitialized {
    public static void main(String[] args) {
        int total;
        total = total + 5;
        System.out.println(total);
    }
}
```

```
Uninitialized.java:4: error: variable total might not have been initialized
        total = total + 5;
                ^
1 error
```

In C, the same code compiles. A local variable that has not been given a value holds whatever happens to be in memory at that moment, and the program may give a different result on every run. Java's strictness is a safeguard here: it moves the error from run time to compile time.

The **off-by-one error** is among the most insidious mistakes. If you write `for i ← 1 to n − 1` when you want the sum from 1 to n, n is skipped. For n = 5 the total comes out as *1 + 2 + 3 + 4 = 10* instead of 15. The program runs, the result looks "plausible," and the error is easily overlooked.

Most of these errors show up with **edge cases**: 0, an empty list, the smallest and largest values, a negative number. A program that works on normal inputs often breaks at the boundaries. For this reason every test set should contain a few edge cases.

Example 4.3.19

A program computes the average of the entered grades with `avg ← total / n`. Which edge cases should be tried, and which error can each reveal?

1. No grades entered (n = 0): division by zero. The program is fixed to print "no grades entered" when n = 0.
2. A single grade (n = 1): the average must be the grade itself. If the loop bound is off by one, the result comes out 0.
3. The smallest and largest grades (0 and 100): the input check must not reject these values by mistake; for example, `grade ≥ 0` should be written rather than `grade > 0`.
4. An invalid grade (−5 or 120): the program must not accept this value.
5. A fractional result (70, 85, 90): the average should come out around 81.67. If there is integer division, 81 comes out.

4.3.7.4 The cost of software bugs

A bug in an assignment results in a low grade. The cost of bugs in real systems can be millions of dollars, even human lives. Three famous examples show the real-world counterparts of the kinds of errors described in this note.

Ariane 5 (1996). On its first flight, the European Space Agency's new rocket Ariane 5 veered off course about 37 seconds after liftoff and broke apart within seconds. According to the inquiry board's report, the cause was a number conversion. The navigation software was converting a 64-bit floating-point horizontal velocity value into a 16-bit signed integer. A 16-bit signed integer can hold at most 32,767. The software had been taken from the slower-climbing Ariane 4, where this limit was never exceeded. Ariane 5's horizontal velocity was much larger, the conversion raised an overflow error, and the navigation system shut down.

Example 4.3.20

What happens in Java if a double with the value 40000.0 is forcibly converted to the 16-bit `short` type?

```
public class Overflow {
    public static void main(String[] args) {
        double speed = 40000.0;
        short narrow = (short) speed;
        System.out.println(narrow);
        System.out.println(Short.MAX_VALUE);
    }
}
```

```
-25536
32767
```

The largest value of the short type is 32,767. 40,000 exceeds this limit. Java gives no error; it keeps only the last 16 bits of the number's binary representation. Because 16 bits can hold $2^{16} = 65{,}536$ different values, the result is $40{,}000 - 65{,}536 = -25{,}536$. A positive speed has silently turned into a negative number. Ariane 5's software was written in the Ada language and, instead of staying silent on overflow, it raised an error. But there was no code to catch and handle that error, and the system shut down. In both cases the root cause is the same: it was never tested whether the value could leave the range the type can hold.

Mars Climate Orbiter (1999). The probe NASA sent to Mars came too close to the planet while entering orbit and was lost. Software at the ground station produced the effect of the thrusters in English units (pound-force seconds). The other software that used this data expected metric units (newton-seconds). One pound-force second is about 4.45 newton-seconds. As a result, the effect of the thrust was computed about 4.45 times too small, and the orbit calculation drifted gradually over the journey of more than nine months. Each of the two programs worked correctly on its own; the bug was in the interface between them.

Therac-25 (1985–1987). The software of a radiotherapy machine used in cancer treatment had a timing bug. When the operator entered settings in a particular order and very quickly, the safety check was skipped. In earlier models hardware interlocks also performed this check; in the Therac-25 this job was left to software alone. In six accidents between 1985 and 1987, patients received radiation doses far above normal; at least three patients died.

All three bugs are of the kind a compiler cannot catch. One is an edge case, one a unit inconsistency, one a timing-dependent logic error. The shared lessons can be gathered in a few points. Edge cases must be tested, and units must be stated clearly in variable names and documentation. Software taken from one system must be retested under the conditions of the new system, and code should be reviewed by someone else.

Common mistake

Not stating the unit in the variable name. From the code you cannot tell whether a variable named `speed` is in km/h or m/s. Names such as `speedKmh` or `speedMs` prevent two programmers from interpreting the same number in different units. The loss of Mars Climate Orbiter shows the value of this small habit.

4.3.7.5 Readable code

Code is written once and read many times. It is read by its own author weeks later, by teammates, and by others who take over maintenance. Readable code both prevents bugs and makes them easier to find. The following two pieces of pseudocode do the same job.

It works but is unreadable:

```
a ← read()
b ← read()
c ← a*b/2
print c
```

The same program, readable:

```
// area of a triangle = base × height / 2
base ← read()
height ← read()
area ← base × height / 2
print "Area: ", area
```

Whoever reads the first program has to work out the formula to understand what c is; `a*b/2` could be a triangle's area, or a mistyped average computation. The second program explains itself. The differences are these:

- **Meaningful names.** area is written instead of c, and base instead of a. The reader understands a variable's role from its name.
- **Consistent indentation and spacing.** The spaces around operations and the regular indentation of nested blocks show the boundaries of the structure at a glance.
- **Named constants instead of magic numbers.** A number embedded in the code whose meaning is not explained is called a **magic number**. Instead of `price * 0.20`, define `VAT_RATE ← 0.20` and write `price * VAT_RATE`. When the rate changes, a single line is corrected.
- **Moving repetition into a function.** If the same formula is written in three places, one fix must be made three times, and one of them will surely be forgotten.

- **Comments.** A comment says not what the code does but why it does it. The comment `// add 1 to x` repeats the code and adds nothing. The comment `// the array starts at 0, add 1 for the ordinal number`, however, explains an intention.

Example 4.3.21

What is the Java version of the readable triangle area program? What are the outputs for base 6, height 5 and base 7, height 3?

```
import java.util.Scanner;

public class TriangleArea {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        // area of a triangle = base × height / 2
        double base = in.nextDouble();
        double height = in.nextDouble();
        double area = base * height / 2;
        System.out.println("Area: " + area);
    }
}
```

```
Area: 15.0
```

```
Area: 10.5
```

6 × 5 / 2 = 15 and *7 × 3 / 2 = 10.5*. Because the variables are double, the second result comes out right with its fractional part. If int had been used, the integer division *21 / 2* would have given 10. The use of a named constant is written in a similar way:

```
public class VatDemo {
    static final double VAT_RATE = 0.20;

    public static void main(String[] args) {
        double price = 250.0;
        double vat = price * VAT_RATE;
        System.out.println("VAT: " + vat);
        System.out.println("Total: " + (price + vat));
    }
}
```

```
VAT: 50.0
Total: 300.0
```

For a product costing 250 TL, the VAT is *250 × 0.20 = 50* TL and the total is 300 TL. If the rate changes, only the `VAT_RATE` line is changed.

Common mistake

Thinking short names make code faster or smaller. The compiler does not carry names into machine code; `c` and `triangleArea` run at the same speed. The only effect of short names is to make the reader's job harder. Short, well-established uses such as the loop counter i are the exception.

4.3.8 Summary

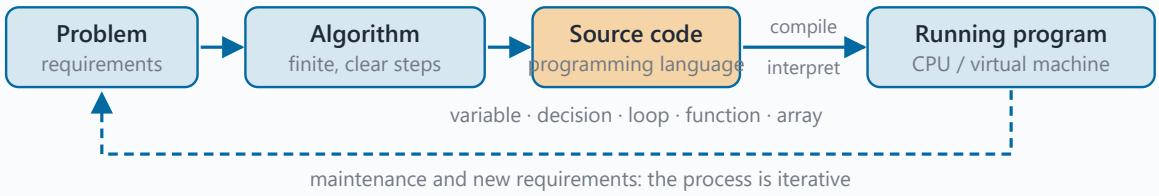


Figure 4.3.33: From problem to running program: the problem becomes an algorithm, and the algorithm becomes source code; the source code runs by compilation or interpretation. Maintenance and new requirements send the process back to the start.

- A **programming language** is a formal language that expresses an algorithm in a form a computer can execute. **Syntax** determines what may be written, and **semantics** what the written text does. A program with correct syntax can do the wrong job.
- **Source code** is the text people read; **machine language** is the binary instructions the processor runs; **assembly** is the symbolic notation of machine instructions. One high-level line often corresponds to several machine instructions.
- Source code is run in three ways. In **compiled** languages (C) the program is translated to machine code beforehand. In **interpreted** languages (Python) commands are evaluated at run time. In **hybrid** languages (Java) the source code is compiled to **bytecode** and the bytecode is executed on a **virtual machine** (the JVM). A **JIT** compiler translates frequently run code into machine code at run time. The compiled/interpreted distinction is a property of the implementation, not of the language.

- There are three kinds of errors. A **compile error** is found during translation, and the program never runs. A **runtime error** stops the program during execution. A **logic error** gives no error message, produces a wrong result, and is found only by testing.
- A compiler works in five phases: lexical analysis (tokens), syntax analysis (syntax tree), semantic analysis (names and types), intermediate code and optimization (e.g. constant folding), and code generation. The first three phases look for errors. The **linker** combines object files and libraries, and the **loader** brings the program into memory and starts it.
- Languages are classified by **level** (low: machine language, assembly; high: C, Java, Python), by **kind** (general-purpose, domain-specific, scripting, markup), and by **paradigm** (imperative, object-oriented, functional). Level is not a measure of quality but of abstraction. The choice of language is determined by the problem area, the platform, libraries, performance, security, and team experience.
- In **static typing** (Java), type errors are found at compile time; in **dynamic typing** (Python), when the line in question runs.
- The path from problem to program: define the problem (input, output, constraints), solve examples by hand, build the algorithm, convert it to code, and test. If testing finds a gap, return to an earlier stage.
- The **software development process** is an iterative process made up of requirements, design, coding, testing, deployment, and maintenance. The **waterfall model** passes through the stages once, in order. The **agile** approach proceeds in short cycles. **Prototyping** makes vague requirements visible. The IDE, compiler, debugger, version control, testing tools, and package manager are the tools of this process.
- **Variables**, **decisions**, **loops**, **functions**, and **arrays** are the basic programming constructs. Operator precedence is as in mathematics. For a loop to end, the body must affect the condition. Every function call opens its own frame on the call stack. Array indices start at 0, and the last index is length − 1.
- Bugs are found by testing with edge cases (0, an empty list, a negative number, the smallest and largest values) and by tracing the program step by step with a trace table. Readable code (meaningful names, named constants, functions, comments that explain "why") both prevents bugs and makes them easier to find.

4.3.9 Key terms

The key terms of this chapter are listed below.

programming language syntax semantics source code machine language machine code

assembly language translator compiler compilation interpreter interpretation

compiled / interpreted bytecode intermediate code virtual machine Java Virtual Machine (JVM)

just-in-time (JIT) compiler compile error syntax error runtime error logic error lexical analysis

token parsing syntax tree semantic analysis code optimization constant folding object file linker linking error loader executable file class loader high-level / low-level language domain-specific language scripting language markup language programming paradigm imperative programming object-oriented programming functional programming static typing dynamic typing waterfall model agile prototyping integrated development environment (IDE) debugger breakpoint package manager version control commit branch merge merge conflict data type operator precedence infinite loop parameter argument call stack array index library application programming interface (API) off-by-one error magic number

---

4.3.10 Test Yourself

1. What kind of error is each of the following Java errors? If it is a compile error, in which phase of the compiler is it caught? (a) `System.out.println("Hello);` (b) Using an undeclared variable `total`. (c) A loop that runs with the condition `i <= n` over an array of n elements. (d) Dividing the total by n − 1 when computing an average.

   Answer

   (a) A compile error; because the unclosed string cannot be turned into a token, it is caught in the lexical analysis phase (javac: "unclosed string literal"). (b) A compile error; whether a name has been declared is checked in the semantic analysis phase ("cannot find symbol"). (c) A runtime error; in the last iteration i = n and `array[n]` is outside the array, so Java throws `ArrayIndexOutOfBoundsException`. (d) A logic error; the program runs with every input but finds the average larger than it is.

2. A three-line Python file reads: `print("A")`, `print(1 / 0)`, `print("B"` (the parenthesis on the last line is unclosed). What appears on the screen when the file is run? What would a simple line-by-line interpreter have shown?

   Answer

   Python parses the whole file before it starts running. It finds the unclosed parenthesis on line 3 and gives only the message `SyntaxError: '(' was never closed`. "A" is not printed, and the division by zero is never reached. A simple line-by-line interpreter would print "A" first, stop at the division by zero on the second line, and never see the syntax error on the third line.

3. Into how many tokens is the line `total = total + 1;` split in lexical analysis? What are the types

of the tokens?

Answer

Six tokens: `total` (identifier), `=` (operator), `total` (identifier), `+` (operator), `1` (constant), `;` (separator). Whitespace is discarded. Because the same name appears twice, it counts as two separate tokens.

4. What does the following pseudocode print? How many times does the body run, and how many times is the condition checked? `s ← 0` ; `i ← 1` ; `while i ≤ 4: s ← s + i × i; i ← i + 2` ; `print s` .

Answer

Iteration 1: *1 ≤ 4* is true; *s = 0 + 1 × 1 = 1*, i = 3. Iteration 2: *3 ≤ 4* is true; *s = 1 + 3 × 3 = 10*, i = 5. Check: *5 ≤ 4* is false, and the loop ends. 10 is printed. The body runs 2 times and the condition is checked 3 times. Because multiplication is done before addition, no parentheses are needed in `s + i × i` .

5. `function f(x): return 2 × x + 1` is defined. What does `print f(f(2))` print? How many frames are opened, and are the two on the stack at the same time?

Answer

The inner call runs first: a frame is opened for f(2), x = 2, *2 × 2 + 1 = 5* is returned, and the frame closes. Then a new frame is opened for the outer call f(5): *2 × 5 + 1 = 11*. 11 is printed. Two frames are opened but they are not on the stack at the same time: the outer call starts after its argument has been computed and the inner frame has closed.

6. In Java, what is s after the line `int s = 17 % 5 * 2 + 9 / 2;` ? Write out the order of computation.

Answer

`%` , `*` , and `/` are at the same precedence level and are done before addition, from left to right. *17 % 5 = 2*, *2 × 2 = 4*. Then *9 / 2* is 4, because it is the quotient of two integers (not 4.5). Finally *4 + 4 = 8*. s = 8.

7. A Python function returns `"a" + 5` if x is greater than 10 and `"small"` otherwise. The program calls `message(3)` and then `message(20)` . When does the error show up? What would happen if

the same logic were written in Java?

Answer

Python parses the file without trouble, because the syntax is correct. `message(3)` returns "small" and it is printed. At the call `message(20)` the faulty line runs for the first time and raises a `TypeError`. In dynamic typing a type error is seen only when the line runs; if x never exceeded 10, the error would never be seen. In Java, in a method whose return type is `String`, `"a" + 5` is string concatenation and is valid: it returns "a5". If the return type were `int`, the compiler would report the type mismatch before the program ever ran.

8. A program to find the smallest element of an array starts with `min ← 0` and compares each element with `if d[i] < min then min ← d[i]`. What is the output for d = [3, 8, 1, 6]? What is the bug, and how is it fixed?

Answer

None of the four comparisons is true: none of 3, 8, 1, and 6 is less than 0. min never changes and the output is 0; the correct result is 1. The bug is in the initial value: 0 is a value from outside the inputs and acts like the smallest value for positive numbers. The fix is `min ← d[0]`. Trace: min = 3; 8 < 3 is false; 1 < 3 is true, min = 1; 6 < 1 is false. The output is 1.

4.3.11 Further reading

- A. B. Downey, C. Mayfield, *Think Java* (2nd ed., O'Reilly, 2019): an introduction to programming with Java; its first chapters cover compilation, interpretation, kinds of errors, and debugging at the level of this note. It can be read free of charge on the Green Tea Press website.
- R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016): teaches programming in Java through science and engineering examples; it has plenty of examples for variables, loops, functions, and arrays.
- R. W. Sebesta, *Concepts of Programming Languages* (12th ed., Pearson, 2019): compares in detail the history of languages, the description of syntax and semantics, type systems, and paradigms.
- A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, *Compilers: Principles, Techniques, and Tools* (2nd ed., Addison-Wesley, 2006): the standard source on the phases of a compiler; for those who want to go deeper into lexical analysis, parsing, and code optimization.

4.3.12 References

1. R. W. Sebesta, *Concepts of Programming Languages*, 12th ed., Pearson, 2019.
2. A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed., Addison-Wesley.

3. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016.

4. A. B. Downey, C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019 (free online version).

5. Oracle, *The Java Tutorials: Getting Started*.

Getting Started with Java

Writing a program is only the first step of running it. The text you write is saved to a file, passes through a compiler, becomes an intermediate code, and runs on a virtual machine. Each step along this path has its own rules and its own error messages. This note begins with the concepts of a program and a programming language. It then describes the origins of Java, the relationship among the JDK, the JRE, and the JVM, and the first Java program line by line. Printing to the console, escape sequences, and string concatenation are worked through with examples. These are followed by the compile-and-run steps, the inside of bytecode, the basic building blocks of the language, and comments. The last sections cover the kinds of errors, how to read error messages, and the preparation to do before coding. All programs shown in the note were compiled and run with JDK 21; the compiler messages and console outputs are taken verbatim from those runs.

LEARNING OUTCOMES

Program
Explain a program as a whole made of input, processing, state, and output; distinguish the syntax, semantics, type system, and execution model of a programming language.

Java
Relate source code, bytecode, and the JVM; explain what the JDK, the JRE, and the JVM contain and what each does.

First program
Write a Java file, compile it with `javac`, and run it with `java`; use the `main` method, command-line arguments, the difference between `print` and `println`, escape sequences, and string concatenation correctly.

Errors
Distinguish syntax, type, runtime, and logic errors; read compiler and JVM messages to find the place and the cause of an error.

5.1 Programs and programming languages

A computer does nothing on its own. A calculator app, a game, and a banking system are all made of commands that tell the computer what to do, step by step. Writing these commands is called **programming**, and the whole of the commands written is called a **program**. The commands are written in an artificial language that both the computer and the human can work with.

5.1.1 What is a program?

Definition 5.1

A **program** is a whole made of commands that a computer executes to perform a particular task. A program can take input, process data, keep intermediate values in memory, and produce output.

The four elements in the definition appear in every program. **Input** is the information given to the program from outside: a number typed at the keyboard, the contents of a file, or a measurement from a sensor. **Processing** is the steps that produce new values from the input: adding, comparing, sorting. **State** is the current condition of the values the program holds in memory while it runs. **Output** is the results the program gives to the outside: a line written to the screen, a saved file, or a lit lamp.

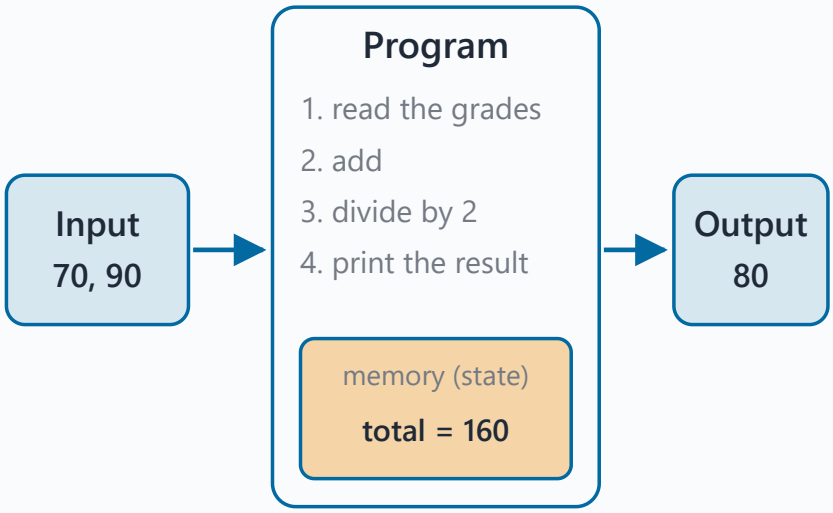


Figure 5.1: A program that finds the average of two grades. The input is 70 and 90. The program reads the grades, adds them, divides by 2, and prints the result. The intermediate value (total = 160) is kept in memory as state; the output is 80.

Figure 5.1 shows these elements on a small example. The program takes two grades: 70 and 90. In the first step the grades are read. In the second step they are added, and the total, 160, is stored in memory. In the third step the total is divided by 2 and 80 results. In the last step 80 is printed to the screen. For the grades 50 and 60, the same program prints 55. The program does not change; only the input changes. This is what makes a program valuable: it is written once and run over and over with different inputs.

Two things determine a program's behavior. The first is the rules of the language it is written in: what each symbol means and how a number is stored. The second is the method the programmer chose, that is, the **algorithm**. In the average example, "first add, then divide" is an algorithm. The same algorithm can be written in different languages. This note uses Java as the language.

5.1.2 The aspects of a programming language

In natural languages such as English, a sentence also has two aspects: whether it is written correctly and what it says. "Me read the book on table" is defective grammatically. "The table read the book" is well formed but makes no sense. In programming languages, two more aspects are added to these two. A programming language is studied from four aspects.

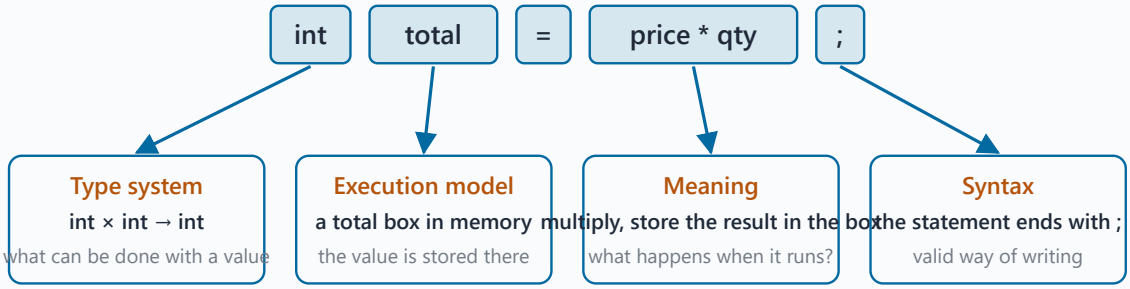


Figure 5.2: The four aspects of the statement `int total = price * qty;`. The syntax requires the statement to end with `;`. The type system says that the product of two `int`s is also an `int`. The meaning is "multiply, store the result in the box." The execution model says that the result is stored in the `total` box in memory.

Definition 5.2

Syntax is the set of rules that determines the valid way of writing a program. **Semantics** defines what happens when a valid piece of program is executed. The **type system** ensures that every value has a type and that only operations appropriate to its type are performed on a value. The **execution model** describes the order in which commands are executed and how values are held in memory.

Figure 5.2 shows these four aspects on a single statement: `int total = price * qty;`. In terms of syntax, the statement starts with a type name, followed by a name, an equals sign, an expression, and a semicolon. If the semicolon is forgotten, the statement is invalid. In terms of the type system,

`price` and `qty` are integers (`int`), and the product of two integers is also an integer. The result can therefore be put into `total` , which has type `int` . In terms of meaning, the statement says "multiply price by qty and write the result to total." In terms of the execution model, a box named `total` is set aside in memory and the result is stored in this box.

In Java, the syntax and type rules are checked before the program is run. This check is done by the **compiler**. A program with a forgotten semicolon, or one that tries to put text into a number box, is never run; the compiler reports the error with a line number. Mistakes concerning meaning, on the other hand, often show up only while the program is running. A program that forgets to divide when computing an average compiles flawlessly but produces a wrong result.

5.2 Source code and tools

The text a programmer writes is called **source code**. Source code is written for humans: it consists of readable words, numbers, and symbols. A processor, however, understands only its own machine instructions. Several tools are used together to turn source code into a running program.

5.2.1 The plain-text source file

A Java program is written to a file with the extension `.java` . This file is called a **source file**. A source file is **plain text**. A plain-text file contains only characters; it has no formatting information such as font, color, bold type, or page layout. A Java file can therefore be opened with Notepad, a code editor such as VS Code, or an integrated development environment. A Word document, on the other hand, is not plain text; it also stores formatting information. If source code is written in Word and saved as `.docx` , the compiler cannot read the file.

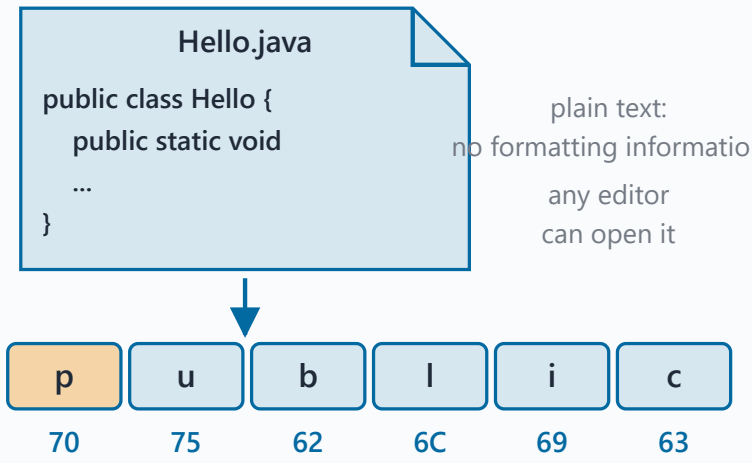


Figure 5.3: `Hello.java` is a plain-text file. On disk each character is stored as one or more bytes: the letters of the word "public" are the bytes 70, 75, 62, 6C, 69, 63 in hexadecimal. ASCII characters take 1 byte in UTF-8,

while letters such as ç, ğ, ş take 2 bytes.

On disk everything is bytes. In a plain-text file, each character is converted to one or more bytes according to a **character encoding**. Today the most common encoding is UTF-8. In UTF-8, the letters of the English alphabet, digits, and punctuation marks take a single byte, as in ASCII. Figure 5.3 shows the bytes of the word "public": the letter p is 70 in hexadecimal, that is, 112 in decimal. The letters ç, ğ, ı, ö, ş, ü, which are specific to Turkish, take two bytes each in UTF-8.

The encoding is an agreement between the programs that write and read the file. If a file is saved as UTF-8 and read with another encoding, non-ASCII letters get corrupted. The word "Dünya" may appear as "DÃ¼nya". The Java compiler also assumes an encoding when it reads a source file. Since JDK 18 the default encoding has been UTF-8. The examples in this note were compiled with JDK 21 without any extra options; words such as "Dünya", "Satır", and "İndirim" came out correctly. If an older JDK is used, the encoding must be stated explicitly to the compiler with the `-encoding UTF-8` option.

Beyond the encoding, two more details should be consistent. The first is line endings: Windows writes a line ending with two characters (CR LF), Linux and macOS with a single character (LF). The Java compiler accepts both. But mixed line endings in the same project cause needless differences in tools that track changes. The second is the file name. In Java, the file name must be the same as the name of the public class inside it: the class `Hello` is written to the file `Hello.java`.

Plain text has one more advantage. **Version control systems** record which line of a file was changed, by whom, and when. Git is the most common example. Because these tools compare line by line, they work best with plain-text files.

5.2.2 Programming tools

Between writing a Java program and running it, it passes through the hands of several tools. Each tool has one specific job.

Tool	Job	Example
Editor	Writing and saving source code	Notepad, VS Code
IDE	Project management, code completion, compiling, running, and debugging	IntelliJ IDEA, Eclipse, NetBeans
`javac`	Compiling source code to bytecode	`javac Hello.java`
`java`	Running a class on the JVM	`java Hello`
`javadoc`	Generating API documentation in HTML from the documentation comments in the source code	`javadoc Hello.java`
Version control	Keeping the history of changes	Git

Table 5.1: Java programming tools and their jobs.

An **editor** is enough to write source code. Unlike ordinary text editors, code editors color keywords, adjust indentation automatically, and show line numbers. An **integrated development environment** (IDE) combines the editor with a compiler, a runner, and a debugger in a single window.

`javac` is the Java compiler; it reads the source code, checks it, and translates it into an intermediate code called **bytecode**. The `java` command starts a Java Virtual Machine and runs the bytecode. `javadoc` generates documentation as web pages from comments written in a special format inside the code. The online documentation of Java's own class library was also produced with this tool.

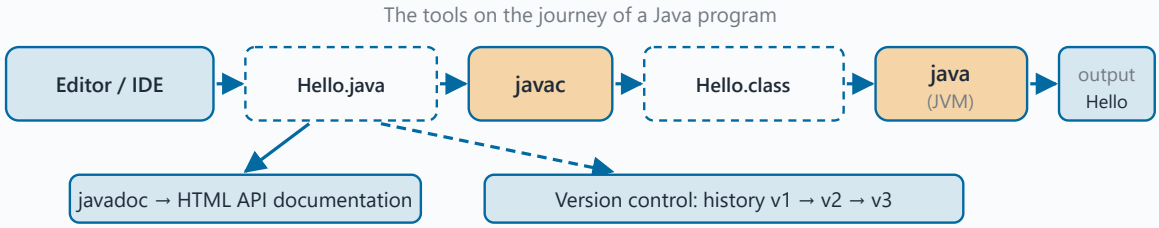


Figure 5.4: The tools on the journey of a Java program. `Hello.java`, written in the editor, is compiled with `javac` into the file `Hello.class`. The `java` command runs this file on the JVM and `Hello` is printed to the console. `javadoc` generates documentation from the source code; version control keeps the history of the source code (v1 → v2 → v3).

Figure 5.4 shows the order of the tools. The point to notice is this: `javadoc` and version control work with the source file, not with the `.class` file. Because the `.class` file can be regenerated at every compilation, it is usually not added to version control. The real product to keep is the source code.

5.3 Java

Java is one of the most widely used programming languages today. We meet it in banking systems, in Android apps, in big-data tools, and in universities' introductory programming courses. The most important reason for this ubiquity is that Java programs can run independently of the operating system and the processor. This section describes how Java was born, where it is used, and what software is needed to run a Java program.

5.3.1 The birth and versions of Java

Java's story begins at Sun Microsystems in 1991. In a project called the "Green project," James Gosling and his team set out to design a language for smart home devices such as television set-top boxes. The language was first called Oak. The home-device market did not attract the expected interest, but in those same years the Web was spreading rapidly. A language that could behave the same way on computers with different operating systems was exactly what the Web needed. The language was released in 1995 under the name Java.

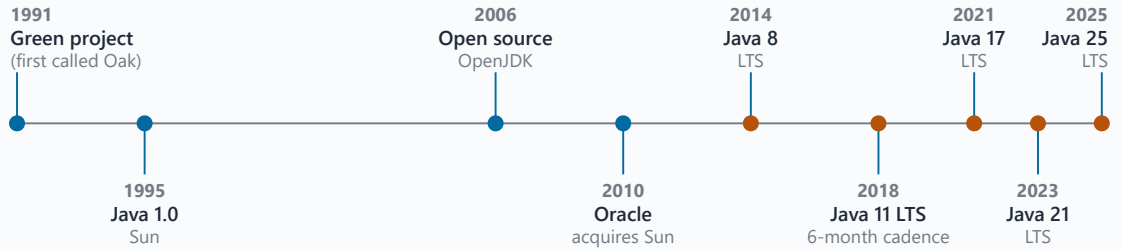


Figure 5.5: The timeline of Java. The Green project (first called Oak) in 1991, Java 1.0 in 1995, the open-source OpenJDK in 2006, and Oracle's acquisition of Sun in 2010. The dots in the accent color are long-term support (LTS) releases: Java 8 (2014), 11 (2018), 17 (2021), 21 (2023), and 25 (2025). Since 2018 new releases come out every six months.

Java's design goal is summed up in a single sentence: **write once, run anywhere**. A Java program is compiled once. The bytecode the compiler produces runs unchanged on Windows, Linux, or macOS, on a Java Virtual Machine written for that system. The programmer does not have to compile separately for each operating system.

Java's other basic features are these:

- **Object-oriented**: Programs are built around classes and objects. Even the smallest Java program is written inside a class.
- **Strong type checking**: Every variable has a type, and the compiler catches type mismatches before the program runs.
- **Automatic memory management**: The memory of objects that are no longer used is reclaimed automatically by **garbage collection**. The programmer does not free memory by hand.
- **A large class library**: There are ready-made classes for jobs such as printing to the screen, reading files, network connections, and date computation.

Java's owner changed in 2010: Oracle bought Sun Microsystems and has developed the language ever since. Java's source code has been open since 2006 and is published in the OpenJDK project. Today, besides Oracle, organizations such as Eclipse Adoptium (Temurin), Amazon (Corretto), and Microsoft offer OpenJDK-based JDK distributions. The examples in this note were run with the Eclipse Adoptium distribution of JDK 21.

Since 2018 a new Java release has come out every six months: in March and September. Each release receives updates for only a few months. Some releases, however, are declared **long-term support** (LTS) releases and receive security updates for years. Organizations and courses therefore prefer LTS releases. Java 8, 11, 17, 21, and 25 are LTS releases. This course uses Java 21.

5.3.2 Where is Java used?

Java's areas of use are very broad. The table below summarizes the best-known ones.

Area	Description
Enterprise servers	Back ends of banking, e-commerce, and government systems; the Spring and Jakarta EE frameworks
Android	Apps are written in Java or Kotlin; the bytecode is converted to a form suited to the Android runtime (ART) on the phone
Big data	Data-processing tools such as Hadoop and Kafka run on the JVM
Development tools	IDEs such as IntelliJ IDEA, Eclipse, and NetBeans are themselves written in Java
Smart cards	The small programs on SIM cards and bank cards run with Java Card
Games	The Java Edition of Minecraft; desktop games and simulations

Table 5.2: The main areas of use of Java.

Some of the examples in the table are the achievement not of the Java language but of the Java Virtual Machine. The JVM does not run only Java. Languages such as Kotlin, Scala, and Groovy are also compiled to the same bytecode and run on the same virtual machine. Most Android apps today are written in Kotlin, but the underlying execution model is the same.

5.3.3 JDK, JRE, and JVM

Three abbreviations appear constantly in Java documents: JDK, JRE, and JVM. The three are nested layers; the outer layer contains the inner one.

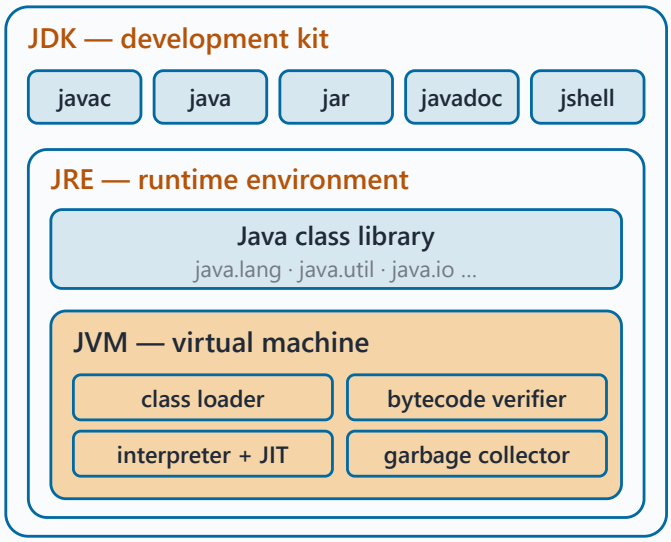


Figure 5.6: The JDK, JRE, and JVM are nested layers. The JDK contains the development tools (javac, java, jar, javadoc, jshell) and the JRE. The JRE contains the Java class library (java.lang, java.util, java.io …) and the JVM. The parts of the JVM are the class loader, the bytecode verifier, the interpreter and JIT compiler, and the garbage collector.

Definition 5.3

The **Java Virtual Machine** (JVM) is the software that executes bytecode. The **JRE** (Java Runtime Environment) is the combination of the JVM and the Java class library: it contains everything needed to run a Java program. The **JDK** (Java Development Kit) adds the development tools (`javac` , `jar` , `javadoc` ...) to the JRE.

In short: JDK = JRE + development tools, JRE = JVM + class library. Figure 5.6 shows this relationship. A user who will only run a ready-made Java program needs just the JRE. A developer who will write and compile programs must install the JDK, because `javac` exists only in the JDK.

This distinction has changed somewhat today. Since Java 11, Oracle no longer offers a separate JRE package; both developers and users install the JDK. Some distributors still offer the JRE separately. Knowing the three layers as concepts remains important, because error messages and documentation use these names.

The five parts of the JVM do their work in turn during the life of a program. The **class loader** reads the `.class` file from disk and brings it into memory. The **bytecode verifier** checks that the loaded code obeys the rules; for example, that an instruction does not try to use a value that is not on the stack. The **interpreter** executes the bytecode instructions one by one. The **JIT compiler** (just-in-time compiler) speeds up the frequently run parts by translating them into the processor's machine code. The **garbage collector** reclaims unused memory as the program runs.

Common mistake

More than one Java may be installed on a computer, and the command line uses the first one it finds. On the computer where this note was prepared, `javac` came from JDK 21 while the `java` on the `PATH` came from an old Java 8 installation. When `Hello.class` , compiled with JDK 21, was run with Java 8, this message was received:

```
Exception in thread "main" java.lang.UnsupportedClassVersionError: Hello has been
compiled by a more recent version of the Java Runtime (class file version 65.0),
this version of the Java Runtime only recognizes class file versions up to 52.0
```

The message is clear: the class file is in version 65 (Java 21) format, while the JVM running it recognizes at most version 52 (Java 8). An old JVM cannot run new bytecode. The fix is to make sure that the outputs of `java -version` and `javac -version` show the same version.

5.4 The first Java program

The traditional first step in learning programming is a program that prints a short greeting to the screen. This program is small, but it contains all the required parts of a Java program. This section first breaks the program into its parts and then traces step by step how the JVM runs it.

5.4.1 The parts of the program

The following program is saved in a file named `Hello.java`:

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

When the program is compiled and run, it writes a single line to the console:

```
> javac Hello.java
> java Hello
Hello
```

In console sessions, lines starting with `>` are commands typed by the user; the lines below are the command's output. There is no line under the `javac` command: when compilation succeeds, the compiler stays silent.



Figure 5.7: The parts of the Hello program. The outer block is the class declaration; the class's name is the same as the file name. The inner block is the `main` method, the entry point of the program. The innermost line is the statement that writes to the console, and it ends with `;`. Each `}` closes the block it opened.

Figure 5.7 shows the program as three nested layers. Let us examine the lines one by one.

1. `public class Hello {` is a **class declaration**. In Java all code lives inside a class. After the `class` keyword comes the name of the class. `public` says that the class is accessible from anywhere. The name of a public class must be exactly the same as the file name: the class `Hello` is written to the file `Hello.java`. The `{` at the end of the line opens the body of the class.
2. `public static void main(String[] args) {` declares the **main method**. A method is a unit of work with a name. When the JVM runs a program, it always starts at the `main` method; for this reason `main` is the program's **entry point**.
3. `System.out.println("Hello");` is the **statement** that does the real work of the program. `System.out` represents the standard output stream, that is, the console. The `println` method writes the text given to it to the console and ends the line. The `"Hello"` in double quotes is a **string**: a text value made of characters. The statement ends with a semicolon.
4. The first `}` closes the body of the `main` method, and the second `}` closes the body of the class.

Each of the words at the head of the `main` method has a job:

Word	Meaning
`public`	The method is accessible from anywhere; the JVM can also call it from outside
`static`	The method belongs to the class; no object needs to be created first to call it
`void`	The method returns no value
`main`	The name of the method; the JVM looks for the entry point by this name
`String[] args`	The method's parameter: an array of strings that carries the command-line arguments

Table 5.3: The words in the declaration of the `main` method.

Each of these words (access modifiers, `static` methods, return types, arrays) is a topic in its own right. For the first programs it is enough to write this line as it is. Still, knowing why each word is there makes it easier to understand the message you get when it is written wrong.

5.4.2 The signature of the main method

A method's name, parameters, and return type make up its **signature**. When the JVM runs a class, it looks for a method with exactly the signature `public static void main(String[] args)`. What is interesting is this: a `main` with a wrongly written signature gives no compile error. To the compiler, a method named `Main`, or one that returns `int`, is an ordinary, valid method. The error appears only when the `java` command tries to start the program.

The table below shows what happens when `main` is written in different forms in class `A` in the file `A.java`. Each version was compiled without errors with `javac A.java` and then run with `java A`.

As written	When run with `java A`
`public static void main(String[] args)`	runs

As written	When run with `java A`
`public static void Main(String[] args)`	`Error: Main method not found in class A, …`
`public void main(String[] args)`	`Error: Main method is not static in class A, …`
`public static int main(String[] args)`	`Error: Main method must return a value of type void in class A, …`
`static void main(String[] args)`	`Error: Main method not found in class A, …`
`public static void main(String a[])`	runs

Table 5.4: Variations of the `main` signature and the result of the `java A` command on JDK 21. Only the first line of each message is shown.

The JVM's message is actually longer and also suggests the correct signature. For example, when `Main` is written with a capital letter, the full message is:

```
Error: Main method not found in class A, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

The JavaFX sentence on the last line is a reminder that JavaFX applications with a graphical interface have another way of starting; it is not used in this course. Three conclusions follow from the table:

- Java is **case-sensitive**: `Main` and `main` are different names.
- `public`, `static`, and `void` are mandatory. When one is missing, the JVM either cannot find the method at all or finds it and rejects it.
- The parameter's name is free: `a` can be written instead of `args`. The old notation `String a[]` is also valid. Still, the Java convention is to write `String[] args`.

In newer versions of Java these rules have been relaxed. From Java 25 on, a `main` method that is not `public` or `static`, or even one with no parameter, can also be launched. This course uses the classic signature that works in Java 21 and in every version.

5.4.3 The program step by step

After the command `java Hello` is typed, several steps pass before `Hello` appears on the screen. Knowing these steps is the basis for understanding runtime errors and stack traces.

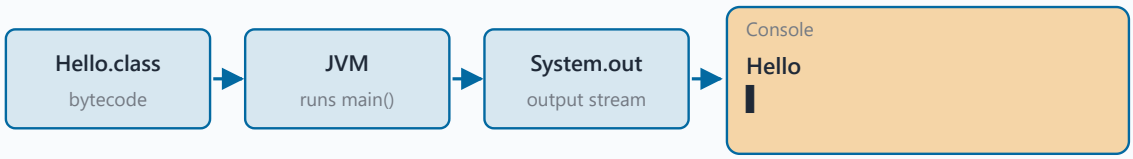


Figure 5.8: The Hello program while running. The JVM loads the bytecode in the `Hello.class` file and runs the `main()` method; the `System.out` output stream carries the text to the console.

Step	Line	What happens	Call stack	Console
1	1	The JVM starts and loads `Hello.class` into memory	empty	
2	2	The JVM calls the `main` method; since no argument was given, `args` is an empty array	`main(args)`	
3	3	The text `"Hello"` is passed to the `println` method	`main(args)` → `println`	
4	3	`println` writes the text and adds a newline; control returns to `main`	`main(args)`	`Hello` ⏎
5	4	The `}` of the `main` block is reached; `main` ends	empty	`Hello` ⏎
6	5	There is no more code to run; the JVM exits	—	`Hello` ⏎

Table 5.5: A step-by-step trace of the `java Hello` command. The call stack shows the methods running at that moment, and the console shows the output so far (⏎ is a newline).

The "call stack" column of the table shows which methods are running at that moment. When one method calls another, the called method is added to the top of the stack; when it finishes, it is removed. In step 3, `main` has called `println`, and both are on the stack. When `println` finishes, only `main` remains. When `main` also finishes, the stack empties and the program ends. In total 1 statement ran and 1 line of output was produced.

5.4.4 Command-line arguments

The `args` parameter of the `main` method is the way to give the program information from the command line. The words written after the class name in the `java` command are called **command-line arguments**. The JVM puts these words into an array of strings and passes it to `main`.

An array is a structure that holds several values of the same type, by sequence number. Each cell in the array has an **index**, and indices start at 0. `args[0]` is the first argument and `args[1]` is the second. `args.length` gives the number of arguments in the array. The following program uses the arguments:

```
public class Greet {
    public static void main(String[] args) {
        System.out.println(args.length);
        System.out.println("Hi " + args[0]);
        System.out.println("Hi " + args[1]);
    }
}
```

The program was run first with two arguments and then with one argument:

```
> java Greet Ali Veli
2
Hi Ali
Hi Veli
> java Greet Ali
1
Hi Ali
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 1 out of bounds
for length 1
        at Greet.main(Greet.java:5)
```



Figure 5.9: The state of the `args` array in two runs. Top, `java Greet Ali Veli`: two cells are filled, `args.length = 2`. Bottom, `java Greet Ali`: only `args[0]` exists, `args[1]` does not; the length is 1.

In the first run the `args` array is `["Ali", "Veli"]` (Figure 5.9, top). Line 3 prints the number of arguments, that is, 2. On line 4 the expression `"Hi " + args[0]` joins two strings side by side and `Hi Ali` is printed. Line 5 prints `Hi Veli` in the same way.

In the second run, both the code and the `.class` file are the same; there is no need to recompile. Only the input changes. The array has a single cell. Lines 3 and 4 print `1` and `Hi Ali`. Line 5 tries to access the nonexistent `args[1]`. In this case the JVM **throws** an **exception**:

`ArrayIndexOutOfBoundsException`, that is, "array index out of bounds." The message says that index 1 lies outside an array of length 1. The line below shows that the error occurred in line 5 of the file `Greet.java`, in the `main` method. The program stops at that point.

The compiler could not have known this error. At compile time it is not known how many arguments the program will be given. The error appears at run time and depends on the input. If no argument is given at all, the error occurs one line earlier, on line 4:

```
> java Greet
0
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds
for length 0
        at Greet.main(Greet.java:4)
```

Example 5.1

The same `Greet` program is run with the command `java Greet Ayşe "Ali Veli"`. What is the output?

Command-line arguments are separated by spaces. Double quotes, however, make the space inside part of the argument. So `"Ali Veli"` is not two arguments but one, and the quotes are not included in the argument. The `args` array is `["Ayşe", "Ali Veli"]`, with length 2:

```
> java Greet Ayşe "Ali Veli"
2
Hi Ayşe
Hi Ali Veli
```

Extra arguments, on the other hand, cause no error. In the command `java Greet 1 2 3` the length of the array is 3. Since the program uses only the first two cells, the third is ignored:

```
> java Greet 1 2 3
3
Hi 1
Hi 2
```

5.4.5 print and println

There are two basic ways to write to the console: `System.out.print` and `System.out.println`. Both write the value given to them. The difference between them appears after the writing is done. `println` (print line) adds a **newline** after writing; the next output starts on the line below. `print` adds no newline; the next output continues on the same line, where it left off.

In the console, the **cursor** determines where the text will appear. It is enough to think of the cursor as the place where the next character will be written. Let us follow the cursor's journey in the following program:

```
public class Greeting {
    public static void main(String[] args) {
        System.out.print("Hello ");
        System.out.println("World");
        System.out.println("Java");
    }
}
```

```
Hello World
Java
```

Line	Statement	Console	Cursor
2	(`main` starts)	▌	line 1, column 1
3	`print("Hello ")`	`Hello` ▌	line 1, column 7
4	`println("World")`	`Hello World` ▌	line 2, column 1
5	`println("Java")`	`Hello World` `Java` ▌	line 3, column 1

Table 5.6: The console and the cursor position after each statement in the `Greeting` program. ▌ shows the cursor; lines and columns start at 1.

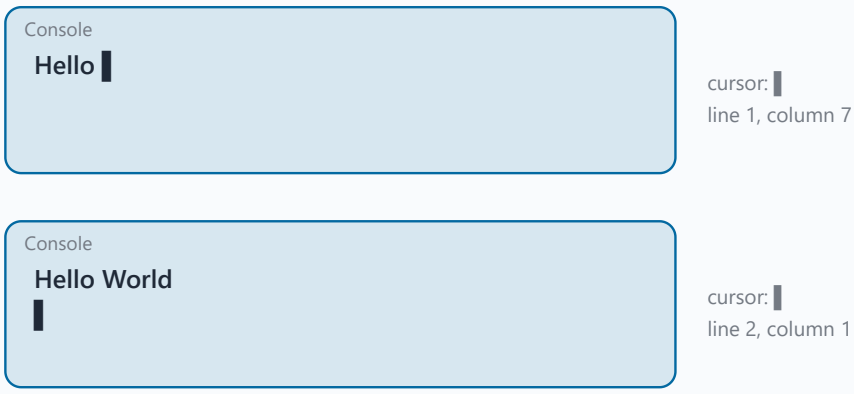


Figure 5.10: Top, after `print("Hello ")` : the cursor is on the same line, right after the space (line 1, column 7). Bottom, after `println("World")` : the line has ended and the cursor is at the start of the next line (line 2, column 1).

The string "Hello " has six characters: five letters and the trailing space. `print` writes these six characters and the cursor stays at column 7 (Figure 5.10). The trailing space matters. Without it "World" would be written right against it, and the output would be `HelloWorld`. `println("World")` continues on the same line and then ends the line. `println("Java")` writes to the new line and ends that line too. Three calls were made, but only two lines were formed, because only `println` ends a line.

`System.out.println()` with no parameter writes only a newline. If the cursor is in the middle of a line, it ends that line; if it is at the start of a line, it produces an empty line. `print`, on the other hand, always needs a value; `System.out.print()` with no parameter is a compile error.

Common mistake

Writing the last output of the program with `print`. The output appears on the screen, but because the line is not ended, the console prompt sticks to the same line: `Result: 80C:\docs>`. Using `println` on the last line of the program prevents this mess.

5.5 Escape sequences and string concatenation

A string begins with a double quote and ends with a double quote. This simple rule raises two questions. First, what do you do if a double quote has to appear inside the text? Second, how do you put characters that cannot be typed directly at the keyboard, such as a newline or a tab, into a string? Java solves both in the same way: escape sequences.

5.5.1 Escape sequences

Definition 5.4

An **escape sequence** is a notation inside a string or character literal that begins with a backslash (`\`) and stands for a single special character. The backslash makes the character that follows "escape" from its usual meaning.

Sequence	Meaning	Example	Output
`\n`	newline (line break)	`"A\nB"`	A, then B on the line below
`\t`	tab	`"Name\tGrade"`	`Name Grade`
`\"`	double quote	`"\"Java\""`	`"Java"`
`\\`	backslash	`"C:\\docs"`	`C:\docs`
`\'`	single quote (inside a character literal)	`'\''`	`'`

Table 5.7: Commonly used escape sequences.

In the source code, `\n` looks like two characters: a backslash and the letter n. The compiler, however, turns it into a single character, the newline character. Neither the backslash nor the n appears on the screen; the line simply ends. In the same way `\"` is a single double-quote character and does not end the string. `\\` gives a single backslash. Because the backslash itself starts escape sequences, the only way to write it is to write it twice.

A single quote causes no problem inside a string: `"Ali's grade"` is written without an escape. The single quote has a special meaning only in `char` literals, which hold a single character. `'A'` is a character literal; the single-quote character itself is written `'\''`.

5.5.2 Tab and newline step by step

The tab character `\t` moves the cursor to the next **tab stop**. In consoles, tab stops are usually every 8 columns: at columns 1, 9, 17, 25 … A tab takes the cursor to the next stop whatever column it is in. That is why values written after words of different lengths line up in the same column. The following program uses this:

```
public class Escape {
    public static void main(String[] args) {
        System.out.println("Name\tGrade");
        System.out.println("Ali\t85");
        System.out.println("\"Java\" rocks");
        System.out.print("Line 1\nLine 2\n");
    }
}
```

```
Name    Grade
Ali     85
"Java" rocks
Line 1
Line 2
```

Line	As written	What happened?
3	`Name`, tab, `Grade`, newline	"Name" occupies columns 1 to 4; the tab moves the cursor to column 9. 4 spaces fall in between; "Grade" starts at column 9.
4	`Ali`, tab, `85`, newline	"Ali" occupies 3 columns; the tab again moves the cursor to column 9. 5 spaces fall in between; 85 lands exactly under the "Grade" heading.
5	`"Java" rocks`, newline	`\"` writes a double quote without ending the string.
6	`Line 1`, newline, `Line 2`, newline	A single `print` produces two lines: the two `\n` in the string end two lines.

Table 5.8: A line-by-line trace of the `Escape` program. Column numbers start at 1.

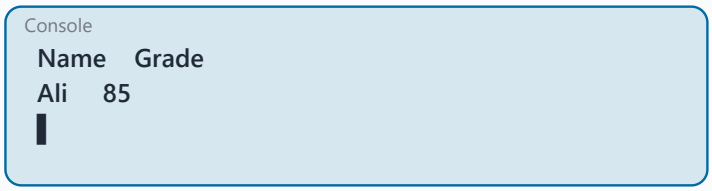


Figure 5.11: The console after the first two lines. Because tabs align to stops 8 columns apart, "Grade" and 85 start in the same column even though "Name" and "Ali" have different lengths.

The `print` call on the last line is worth noting. `print` itself adds no newline, but the `\n` characters inside the string end the lines. So `System.out.print("Line 1\nLine 2\n")` and two separate `println` calls give the same output. One detail: the newline that `println` adds varies by operating system. On Windows, `println` writes two characters (`\r\n`), while `\n` is a single character everywhere. In the console the two look the same.

Alignment with tabs has a limit. If a word is 8 characters or longer, the tab moves the cursor to the next stop, column 17, and the alignment breaks. That is why tabs suit simple tables with short headings.

Example 5.2

How do you write a program that prints a student's term courses as an aligned table with their credit and ECTS values, and adds the totals below it?

Each row is written with a `println`, and the columns are separated with `\t`. The totals are written on two lines with a single `print` and two `\n`:

```
public class Card {
    public static void main(String[] args) {
        System.out.println("Course\tCredits\tECTS");
        System.out.println("Physics\t4\t6");
        System.out.println("Java\t3\t5");
        System.out.print("Total credits: 7\nTotal ECTS: 11\n");
    }
}
```

```
Course  Credits ECTS
Physics 4       6
Java    3       5
Total credits: 7
Total ECTS: 11
```

"Course" has 6 characters, "Physics" 7, and "Java" 4. Since all three are shorter than 8, the first tab moves the cursor to column 9 on every line, and the credit values line up. The second tab moves to column 17; the ECTS values are aligned too. Because "Physics" has 7 characters, only 1 space follows it.

5.5.3 Escape sequence pitfalls

The backslash causes the most trouble when writing Windows file paths. Windows separates folders with a backslash: `C:\data`. If this path is written into a string as it is, the compiler takes `\d` as the start of an escape sequence:

```
public class BadPath {
    public static void main(String[] args) {
        System.out.println("C:\data");
    }
}
```

```
BadPath.java:3: error: illegal escape character
        System.out.println("C:\data");
                           ^
1 error
```

There is no escape sequence `\d`. The compiler says `illegal escape character` and points at the problem with `^`. The program does not compile. A more insidious case is when the letter after the backslash forms a valid escape sequence:



Figure 5.12: The string `"C:\new"` looks like 6 characters in the source but is 5 characters in memory: C, :, newline, e, w. The screen shows "C:" and, on the line below, "ew".

Statement	Result
`System.out.println("C:\data");`	compile error: `illegal escape character`; there is no escape sequence `\d`
`System.out.println("C:\new");`	compiles but prints two lines: `C:` and `ew`
`System.out.println("C:\\data");`	`C:\data`: the backslash is written twice
`System.out.println("Ali's grade");`	`Ali's grade`: a single quote inside a string is written without an escape
`System.out.println("Caf\u00E9");`	`Café`: `\u` and four hexadecimal digits give a Unicode character

Statement	Result
`System.out.println("C:/data");`	`C:/data` : Java's file operations accept a forward slash on Windows too
`System.out.println("A\\nB");`	`A\nB` (one line): `\\` gives one backslash, and the n that follows is an ordinary letter

Table 5.9: Escape sequence pitfalls and their results on JDK 21.

Run together in a single program, in the order of the two tables, all the examples that can be compiled give this output:

```
public class Escapes {
    public static void main(String[] args) {
        System.out.println("A\nB");
        System.out.println("Name\tGrade");
        System.out.println("\"Java\"");
        System.out.println("C:\\docs");
        System.out.println('\'');
        System.out.println("C:\new");
        System.out.println("C:\\data");
        System.out.println("Ali's grade");
        System.out.println("Caf\u00E9");
        System.out.println("C:/data");
        System.out.println("A\\nB");
    }
}
```

```
A
B
Name    Grade
"Java"
C:\docs
'
C:
ew
C:\data
Ali's grade
Café
C:/data
A\nB
```

Lines 7 and 8 of the output come from the statement `"C:\new"` (Figure 5.12). Because the program runs without error, this mistake is easy to miss: the compiler does not warn; only the output is odd. The last line shows the opposite: `"A\\nB"` produces no newline and writes a backslash and the letter n to the screen.

The notation that starts with `\u` is called a **Unicode escape**. The four hexadecimal digits that follow are the Unicode number of the character. The number of the letter é is 00E9 in hexadecimal. Unicode escapes are used to write characters that are not on the keyboard or that risk corruption because of the file encoding:

```
public class Unicode {
    public static void main(String[] args) {
        System.out.println("Café");
        System.out.println("İstanbul");
        System.out.println("Zürich");
    }
}
```

```
Café
İstanbul
Zürich
```

Common mistake

Writing a Windows path with a single backslash. Whether `"C:\docs\notes"` compiles depends on the letters: because `\d` is invalid, you get a compile error. `"C:\temp\new"`, on the other hand, compiles, but `\t` becomes a tab and `\n` a newline. The rule is simple: every backslash in a string is written as `\\`, or the path is written with forward slashes as `"C:/temp/new"`.

5.5.4 String concatenation

The `+` sign does two different jobs in Java. Between two numbers it **adds**: the result of `40 + 45` is 85. If at least one operand is a string, it **concatenates**: the other operand is converted to text and appended. The result of `"Score: " + 85` is the string `"Score: 85"`. This second job is called **string concatenation**.

When an expression contains several `+` signs, two rules determine the result:

1. Operations at the same precedence are performed **from left to right**. The expression `a + b + c` is evaluated as `(a + b) + c`.
2. `*` and `/` are always performed before `+`. Parentheses come before everything.

At each `+` step the question "is one of the operands a string?" is asked afresh. If the answer is yes, concatenation is done; if no, addition. Once a string has appeared, all the `+` signs to its right are concatenations, because the intermediate result on the left is now a string.

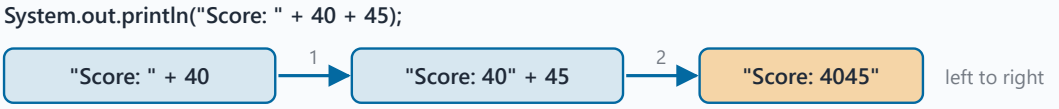


Figure 5.13: `"Score: " + 40 + 45` is evaluated left to right with two concatenations: first `"Score: " + 40` → `"Score: 40"`, then `"Score: 40" + 45` → `"Score: 4045"`.

```
public class Concat {
    public static void main(String[] args) {
        System.out.println(40 + 45 + " points");
        System.out.println("Score: " + 40 + 45);
        System.out.println("Score: " + (40 + 45));
        System.out.println("Avg: " + 85 / 2);
        System.out.println(5 + 3);
        System.out.println("5 + 3");
    }
}
```

```
85 points
Score: 4045
Score: 85
Avg: 42
8
5 + 3
```

Expression	Steps	Output
`40 + 45 + " points"`	`40 + 45` → 85 (two `int`s: addition); `85 + " points"` → `"85 points"`	`85 points`
`"Score: " + 40 + 45`	`"Score: " + 40` → `"Score: 40"`; `"Score: 40" + 45` → `"Score: 4045"`	`Score: 4045`
`"Score: " + (40 + 45)`	parentheses first: `40 + 45` → 85; `"Score: " + 85` → `"Score: 85"`	`Score: 85`
`"Avg: " + 85 / 2`	`/` first: `85 / 2` → 42 (when two integers are divided, the fraction is discarded); `"Avg: " + 42`	`Avg: 42`
`5 + 3`	two `int`s: addition	`8`
`"5 + 3"`	everything inside quotes is text; no computation is done	`5 + 3`

Table 5.10: Step-by-step evaluation of expressions containing `+`.

The fourth row of the table needs separate attention. In `85 / 2`, both operands are integers, so the result is an integer too. The fractional part is not rounded but discarded: the result is 42, not 42.5. Note that here `/` is performed before `+` and that the result is appended to the string as 42.

Common mistake

Writing a sum with a description but without parentheses. The statement `System.out.println("Sum: " + a + b);` prints `Sum: 4045` for a = 40 and b = 45, because the leftmost operand is a string and both numbers are appended to the text. If the sum itself is wanted, `"Sum: " + (a + b)` must be written.

5.6 The structure of a Java file

The Hello program consisted of a single class. In real programs, two more kinds of declarations can appear before the class at the top of a file: `package` and `import`. The parts of a Java file are written in a fixed order.

```
package school;

import java.util.Scanner;

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

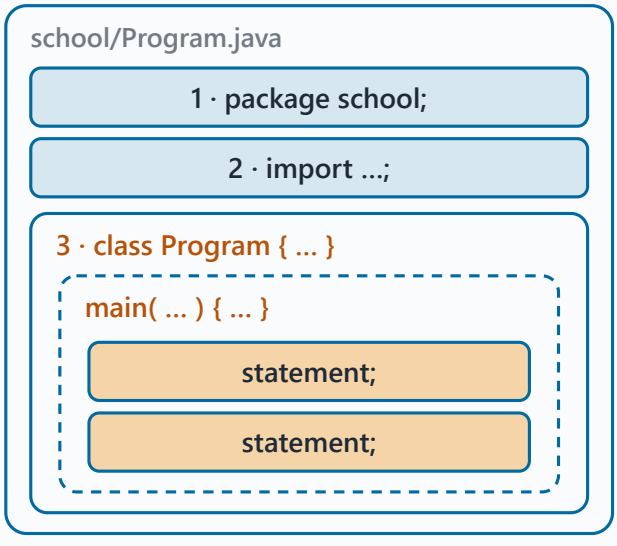


Figure 5.14: The nested structure of the file `school/Program.java`: 1. the `package` declaration, 2. the `import` declarations, 3. the class. Inside the class is the `main` method, and inside the method are the statements.

Figure 5.14 shows the structure. The order is as follows:

1. `package school;` says which **package** the class belongs to. A package is a namespace that keeps related classes together; it is similar to folders on a computer. If there is a package declaration, it must be the first statement in the file, and the file lives in a folder with the same name as the package: `school/Program.java`.
2. `import java.util.Scanner;` makes it possible to use a class from another package by its short name. Without this line the class would have to be written everywhere with its full name, `java.util.Scanner`. `import` declarations come after the package declaration and before the class. In this example `Scanner` is not used; it is written only to show the place. An unused `import` gives no error.
3. `public class Program { … }` is the class itself. The name of a public class is the same as the file name.

Note that no `import` is needed for the `System` class. The most basic classes such as `System` and `String` are in the `java.lang` package, and this package is added to every file automatically.

A class with a package is compiled and run from the folder one level above the package folder. When running, the full name of the class, including the package name, is written:

```
> javac school/Program.java
> java school.Program
Hello
> java Program
Error: Could not find or load main class Program
Caused by: java.lang.ClassNotFoundException: Program
```

When only `Program` is typed in the last command, the JVM looks for a `Program` class with no package and cannot find it. The real name of the class is `school.Program`. The first programs use no package; the classes live in the default, unnamed package. IDEs, however, often add a package declaration when creating a new project. This detail must not be forgotten when a file written in an IDE is compiled on the command line.

5.7 Compiling and running

A Java program does not run the moment it is written. It is first compiled and then run. These two steps are done with two separate commands and involve two kinds of files: the compiler reads the `.java` file, while the JVM runs the `.class` file. Understanding this distinction well prevents most of the problems beginners run into most often.

5.7.1 The compilation pipeline

Definition 5.5

Compilation is checking the source code as a whole and translating it into another language. In Java, the compiler `javac` translates the source code into a processor-independent intermediate code called **bytecode**. Bytecode is stored in files with the extension `.class` and runs on the Java Virtual Machine.

There are five steps from writing a Java program to seeing its output. Let us call all of these steps together the **compilation pipeline**:

1. **Write.** The source code is written in an editor and saved as `Hello.java`. The file name must be the same as the name of the public class.
2. **Compile.** The command `javac Hello.java` runs the compiler. The compiler reads the whole file before the program has even started to run; it checks the syntax and the types.
3. **Bytecode.** If compilation succeeds, the file `Hello.class` appears in the same folder. This file is not plain text but a binary file. It is the same on Windows, Linux, and macOS.
4. **Run.** The command `java Hello` starts the JVM. This command is given the class name, not the file name; the `.class` extension is not written. The JVM's class loader brings the `.class` file into memory, the verifier checks the bytecode, and then `main` is called.
5. **Output.** The JVM interprets the bytecode; it translates the frequently run parts into machine code with the JIT compiler. `println` writes `Hello` to the console.

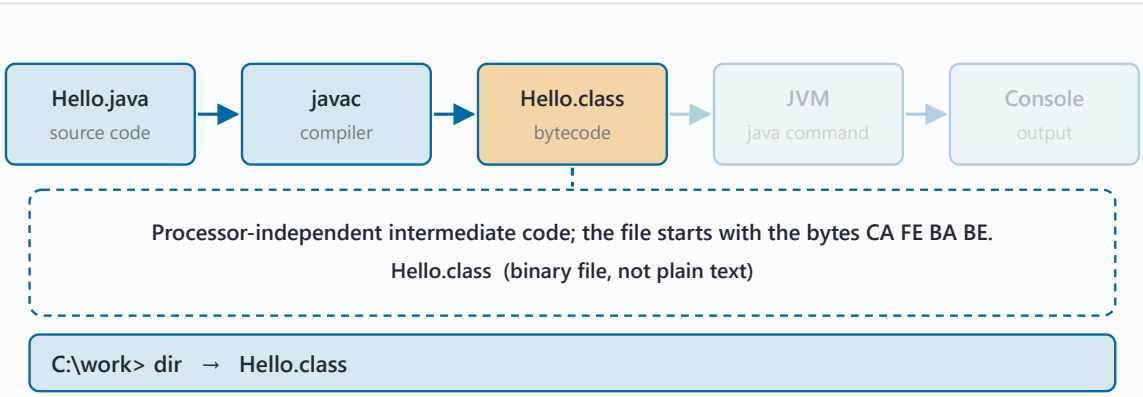


Figure 5.15: The third step of the compilation pipeline. When `javac` succeeds, `Hello.class` appears in the folder. This file is a processor-independent intermediate code, and its first four bytes are CA FE BA BE in hexadecimal.

Every `.class` file begins with the same four bytes: `CA FE BA BE` in hexadecimal. This "magic number" lets the JVM recognize that the file really is a Java class file. The first eight bytes of the `Hello.class` file compiled for this note are `ca fe ba be 00 00 00 41`. The last two bytes are the

class file's version number: 41 in hexadecimal is 65 in decimal and denotes Java 21. This is the "class file version 65.0" mentioned when an old JVM rejects a new class file.

If the compiler finds an error, the pipeline stops at the second step:

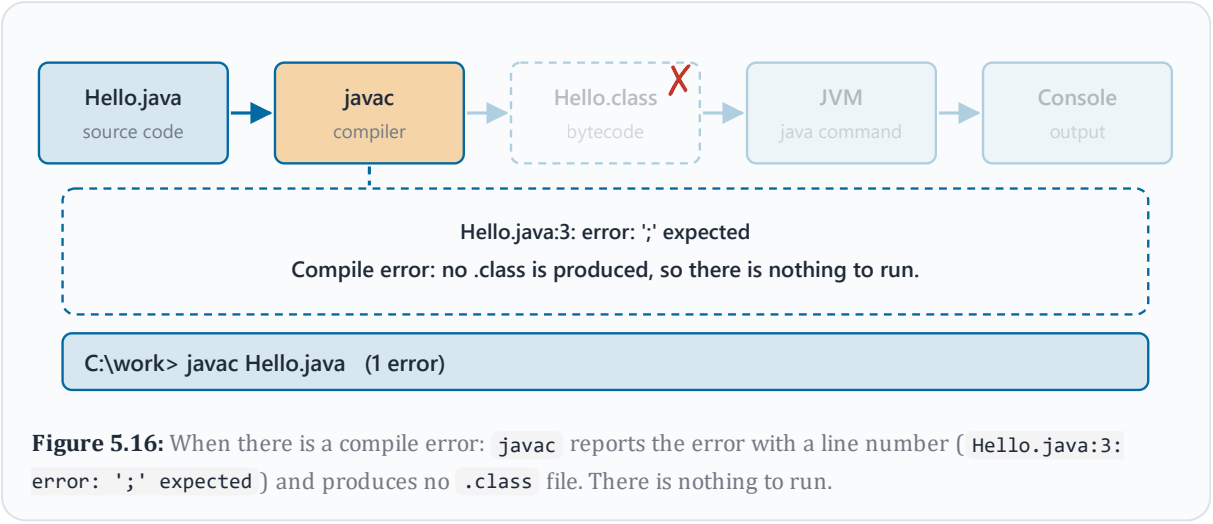


Figure 5.16: When there is a compile error: `javac` reports the error with a line number (`Hello.java:3: error: ';' expected`) and produces no `.class` file. There is nothing to run.

A program with a compile error does not run at all; not even part of it runs. The compiler checks the whole file, and if it finds even a single error it produces no `.class` . This is an important difference from interpreted languages that run line by line. In a language such as Python, the lines before the faulty line may run. In Java, an error on line 50 means line 1 will not run either.

5.7.2 Installing the JDK and the commands

To write Java programs you first install a JDK. After installation, a few basic commands are used on the command line:

Command	What it does
`java -version`	shows the installed Java version
`javac -version`	shows the compiler version (`javac` exists only in the JDK)
`javac Hello.java`	compiles: produces `Hello.class`
`java Hello`	runs (no extension is written)
`java Hello.java`	Java 11 and later: compiles and runs a single-file program, leaving no `.class` file

Table 5.11: The basic JDK commands.

After installation, the first thing to do is to run the two version commands and check that they show the same version. On the computer used for this note, the output of the JDK 21 commands is:

```
> 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
```

The usage in the last row of the table is practical for quick experiments. If a `.java` file is given to the `java` command, the JVM compiles the file in memory and runs it immediately:

```
> java Hello.java
Hello
```

This way works for single-file programs and leaves no `.class` file on disk. To show that compiling and running are two separate steps, this note uses the two-step way.

PATH and JAVA_HOME. When `javac` is typed on the command line, the operating system looks for a program of that name in the folders listed in the `PATH` **environment variable**. If the JDK's `bin` folder is not on the `PATH`, you get a "command not found" error. The fix is to add the JDK's `bin` folder, for example `C:\Program Files\Eclipse Adoptium\jdk-21.0.3.9-hotspot\bin`, to the `PATH`. The `JAVA_HOME` environment variable points to the JDK's main folder; build tools and IDEs look at this variable to find Java.

Non-ASCII letters in the Windows console. Turkish letters can get corrupted in two places. The first is at compile time: if the compiler reads the file with the wrong encoding, the letters are corrupted before they even enter the bytecode. In JDK 18 and later the compiler's default encoding is UTF-8; in earlier versions you have to write `javac -encoding UTF-8 Hello.java`. The second is the console itself. If the program runs correctly but the letters look corrupted on the screen, the console's code page can be switched to UTF-8: first run the command `chcp 65001`.

5.7.3 Compile, change, recompile

One of the things that surprises beginners most is this: the source code is changed, the program is run, but the old output appears. The reason is that the `java` command never looks at the source code. The JVM runs only the bytecode in the `.class` file. In the following session, the folder initially contains only `Hello.java`, and the program prints `Hello`.

```
> java Hello
Error: Could not find or load main class Hello
Caused by: java.lang.ClassNotFoundException: Hello
> javac Hello.java
> java Hello.class
Error: Could not find or load main class Hello.class
Caused by: java.lang.ClassNotFoundException: Hello.class
> java Hello
Hello
```

Three separate situations appear in this part. In the first command there is no `Hello.class` yet; the JVM cannot find a class named `Hello`. The `Caused by` line gives the underlying cause of the error: `ClassNotFoundException`, that is, the class was not found. Compilation then succeeds silently. In the third command, `java` is given a file name. The JVM looks for a class named "Hello.class". Because class names cannot contain a dot in Java, it interprets this as a class named `class` in a package named `Hello` and cannot find it. The correct command, without the extension, runs the program.

Now in the editor the text `"Hello"` is changed to `"Hi"`, the file is saved, and the session continues:

```
> java Hello
Hello
> javac Hello.java
> java Hello
Hi
```

Step	Command or action	`Hello.java`	`Hello.class`	Output
1	`javac Hello.java`	`println("Hello")`	prints "Hello" (new)	—
2	`java Hello`	`println("Hello")`	prints "Hello"	`Hello`
3	edit and save	`println("Hi")`	prints "Hello" (old!)	—
4	`java Hello`	`println("Hi")`	prints "Hello" (old!)	`Hello`
5	`javac Hello.java`	`println("Hi")`	prints "Hi" (new)	—
6	`java Hello`	`println("Hi")`	prints "Hi"	`Hi`

Table 5.12: The contents of the two files over the session. Although the source has changed, the `.class` is updated only when it is recompiled.

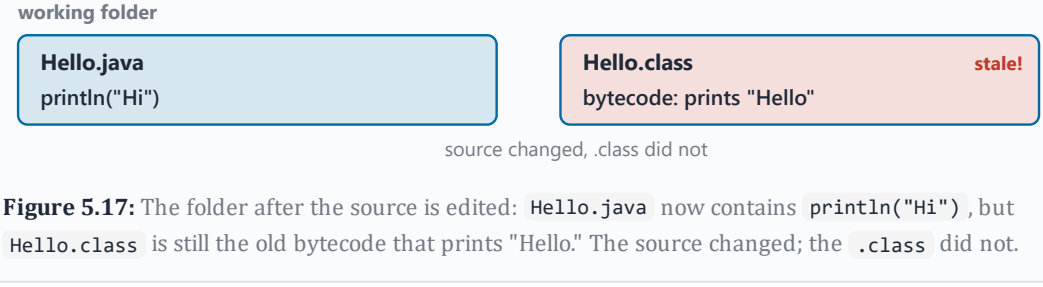


Figure 5.17: The folder after the source is edited: `Hello.java` now contains `println("Hi")`, but `Hello.class` is still the old bytecode that prints "Hello." The source changed; the `.class` did not.

Table 5.12 and Figure 5.17 show the same result: the `.class` file is not updated by itself. In the fourth step the output is still `Hello`. You must recompile after every change to the source code.

Letter case in the class name also matters. Windows file names do not distinguish case, but Java does. With the command `java hello`, Windows finds the file `Hello.class`, but the JVM sees that the name of the class inside the file is `Hello`, not `hello`, and rejects it:

```
> java hello
Error: Could not find or load main class hello
Caused by: java.lang.NoClassDefFoundError: hello (wrong name: Hello)
```

The phrase `wrong name: Hello` in the message means "wrong name; the correct one is Hello."

5.7.4 Working with an IDE

Compiling and running on the command line shows plainly what happens behind the scenes. In daily work, however, most programmers use an IDE. The IDE hides the same steps behind a button.

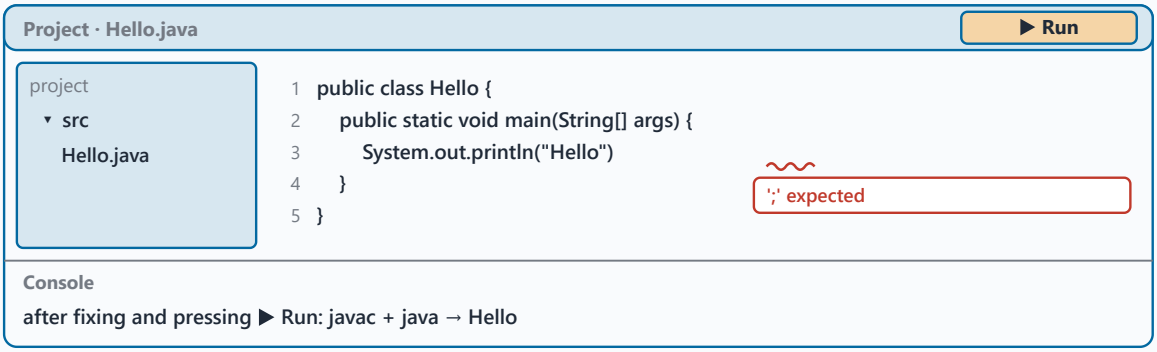


Figure 5.18: An IDE window. On the left the project tree, in the middle the editor, at the bottom the console. The missing semicolon on line 3 is marked with a red wavy line before compiling, and the compiler's message (`';' expected`) is shown. When the error is fixed and the ▶ Run button is pressed, the IDE first compiles and then runs.

The main conveniences an IDE provides are these:

- **The ▶ Run button** performs the compile and run steps back to back. Changed files are recompiled first; for that reason the stale-`.class` problem does not arise.
- **Instant error display**: the IDE compiles in the background while you type. Compile errors are marked with a red underline; when the pointer is over it, the compiler's message appears.
- **Code completion**: when you type `System.out.`, the methods that can be used are listed. Misspelled method names become rarer this way.
- **Automatic indentation** and **renaming**: when a variable's name is changed, it changes everywhere it occurs.

- **Debugger**: stops the program at a **breakpoint**, shows the current values of the variables, and steps through the program line by line.

The common Java IDEs are IntelliJ IDEA, Eclipse, and NetBeans. VS Code does the same jobs with Java extensions. An IDE is a great convenience, but knowing what happens behind the scenes is still necessary. On servers or over remote connections there is often no IDE; there is only the command line. The messages an IDE shows are also `javac` 's messages. Someone who knows the compile steps on the command line can read these messages in any environment.

5.8 Bytecode and the JVM

Java's "compile once, run anywhere" principle is possible thanks to bytecode. This section explains why bytecode is processor-independent, what is inside a `.class` file, and how the JVM executes bytecode.

5.8.1 Compile once, run anywhere

In languages such as C, the compiler translates source code directly into the machine code of a particular processor. A program compiled for a Windows computer with an Intel processor does not run on a Mac with an ARM processor; it has to be compiled separately for that one. The Java compiler, however, produces the machine code of no real processor. It produces the instructions of an imaginary processor that does not exist, namely the Java Virtual Machine. These instructions are called **bytecode**.

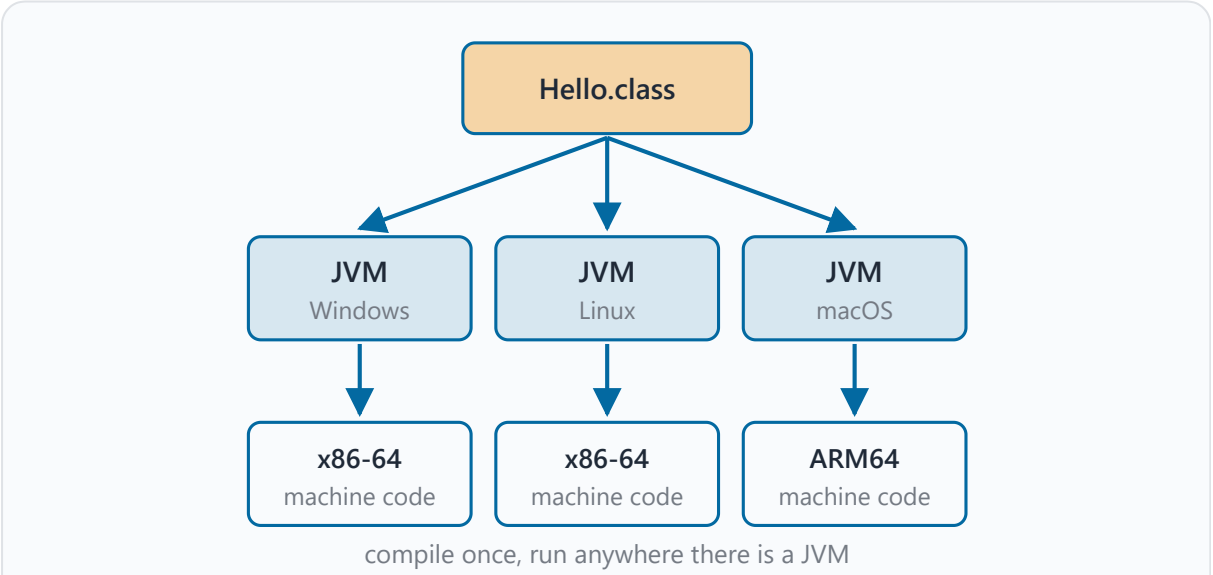


Figure 5.19: The same `Hello.class` file runs on the JVMs on Windows (x86-64), Linux (x86-64), and macOS (ARM64). Each JVM translates the bytecode into the machine code of its own processor.

What makes the imaginary processor real is the JVM written separately for each platform (Figure 5.19). The Windows JVM and the macOS JVM read the same bytecode. Each runs it on its own processor. Two ways are used for this. The interpreter reads the bytecode instructions one by one and executes them. The JIT compiler translates the frequently run parts, for example a loop that runs thousands of times, into that processor's machine code and runs the machine code directly on later iterations. In this way Java programs are both portable and fast.

Class loading, bytecode verification, and garbage collection are also parts of the runtime. The verifier is especially important. A `.class` file may have been downloaded from the network or may be corrupted. The verifier checks, before running, that the code obeys the JVM's rules.

5.8.2 Inside bytecode: javap

A `.class` file is a binary file; if it is opened with Notepad, meaningless characters appear. The `javap` tool that comes with the JDK is used to read the instructions inside it. The `-c` option lists the bytecode instructions of each method in readable form:

```
> javap -c Hello
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #7                  // Field
java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #13                 // String Hello
       5: invokevirtual #15                 // Method java/io/PrintStream.println:
(Ljava/lang/String;)V
       8: return
}
```

The output has two methods. The first is `public Hello();`, that is, the class's **constructor**. No such method was written in the source code. If no constructor is written for a class, the compiler adds an empty one itself. The second method is `main`, and it consists of four instructions:

Position	Command	What it does
0	`getstatic`	pushes the reference to the `System.out` object onto the operand stack
3	`ldc`	pushes the constant `"Hello"` onto the operand stack
5	`invokevirtual`	takes two values from the stack and calls `println`

Position	Command	What it does
8	`return`	ends `main`

Table 5.13: The bytecode instructions of the `Hello.main` method.

The numbers to the left of the instructions are the instruction's **byte position** within the method. The opcode of each instruction is a single byte; the name "bytecode" comes from this. Some instructions are followed by operand bytes. `getstatic` takes 3 bytes (1 opcode, 2 operand), so the next instruction is at position 3. `ldc` takes 2 bytes, and `invokevirtual` starts at position 5. It also takes 3 bytes; `return` is at position 8. Markers such as `#7` and `#13` are entry numbers in the class file's **constant pool**. `javap` writes what each entry is in the comment on the right: `#13` is the string "Hello."

The JVM is a **stack machine**. Real processors hold values in small memory cells called registers when they compute. The JVM instead puts values on an **operand stack** and performs operations on the values at the top of the stack. A stack is like plates piled on top of each other: the value put on last is taken off first. Not using registers keeps bytecode processor-independent, because every processor has a different number of registers.

Comments and whitespace in the source code do not enter the bytecode. A Hello program with plenty of comments (5.10) compiles to the same four instructions.

5.8.3 The JVM executes bytecode

To see how the stack machine works, let us trace a slightly longer program. The program adds two numbers and prints the result:

```
public class Sum {
    public static void main(String[] args) {
        int a = 3;
        int b = 4;
        int c = a + b;
        System.out.println(c);
    }
}
```

```
7
```

The command `javap -c Sum` gives these instructions for the `main` method:

```
 0: iconst_3
 1: istore_1
 2: iconst_4
 3: istore_2
 4: iload_1
 5: iload_2
 6: iadd
 7: istore_3
 8: getstatic     // System.out
11: iload_3
12: invokevirtual // println(int)
15: return
```

The four lines in the source have turned into 12 bytecode instructions. The names of the instructions follow a pattern. The leading `i` stands for `int`. `iconst_3` pushes the constant 3 onto the stack. `istore_1` takes the value at the top of the stack and writes it into local variable number 1. `iload_1` copies the value of local variable number 1 onto the stack. `iadd` takes two values from the stack, adds them, and pushes the result onto the stack.

The method's **local variables** are held in numbered slots. Slot 0 holds `main`'s parameter `args`. The following slots are assigned in the order of the source code: number 1 is `a`, number 2 is `b`, number 3 is `c`. The table below shows the slots and the operand stack after each instruction.

Position	Command	1 (a)	2 (b)	3 (c)	Operand stack	Console
0	`iconst_3`	—	—	—	[3]	
1	`istore_1`	3	—	—	[ ]	
2	`iconst_4`	3	—	—	[4]	
3	`istore_2`	3	4	—	[ ]	
4	`iload_1`	3	4	—	[3]	
5	`iload_2`	3	4	—	[3, 4]	
6	`iadd`	3	4	—	[7]	
7	`istore_3`	3	4	7	[ ]	
8	`getstatic`	3	4	7	[System.out]	
11	`iload_3`	3	4	7	[System.out, 7]	
12	`invokevirtual`	3	4	7	[ ]	`7`
15	`return`	3	4	7	[ ]	`7`

Table 5.14: The trace table of the `Sum.main` bytecode. The stack is written from left to right; the rightmost value is at the top. — marks an empty slot.

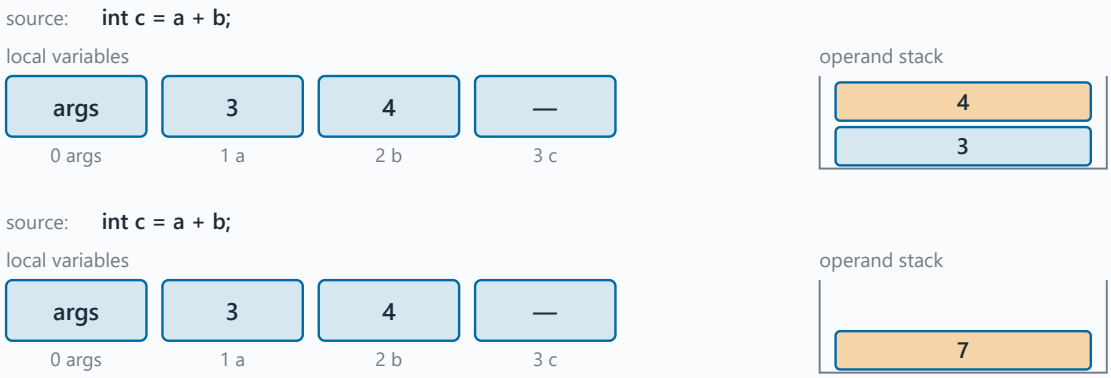


Figure 5.20: Before and after the `iadd` instruction. Top, after `iload_1` and `iload_2`: the stack holds 3 and 4. Bottom, after `iadd`: the two values were taken off and their sum, 7, was pushed. The local variables do not change in the meantime.

A pattern stands out in the trace. The line `int c = a + b;` becomes four instructions: two loads, one addition, one store. The addition instruction `iadd` knows nothing of variable names; it works only with the two values at the top of the stack. Bringing values onto the stack is the job of `iload` and taking them from the stack to a variable is the job of `istore` (Figure 5.20).

The line `System.out.println(c);` follows the same pattern. First the object on which `println` will be called (`System.out`), then the value to be printed (7), are pushed onto the stack. `invokevirtual` takes both off the stack and calls the method `println(int)`. 7 is written to the console. `main` ends with `return`. The method's frame is removed, and the local variables and the operand stack vanish with the frame.

The gaps in the instruction positions can also be explained. Every instruction from 0 to 7 is a single byte, because instructions such as `iconst_3` and `istore_1` carry their operands in their names. Because `getstatic` takes 3 bytes, the next instruction is at position 11. `iload_3` is 1 byte and `invokevirtual` is 3 bytes, so `return` falls at position 15.

Example 5.3

In the `Sum` program, how does the bytecode change if 5 is written instead of 3, 2 instead of 4, and multiplication is used instead of addition?

```
public class Product {
    public static void main(String[] args) {
        int a = 5;
        int b = 2;
        int c = a * b;
        System.out.println(c);
    }
}
```

The structure of the instructions stays the same; only the constants and the operation change. `iconst_5` replaces `iconst_3`, `iconst_2` replaces `iconst_4`, and the multiplication instruction `imul` replaces `iadd`. `javap -c Product` confirms this prediction:

```
public static void main(java.lang.String[]);
  Code:
     0: iconst_5
     1: istore_1
     2: iconst_2
     3: istore_2
     4: iload_1
     5: iload_2
     6: imul
     7: istore_3
     8: getstatic     #7                  // Field
java/lang/System.out:Ljava/io/PrintStream;
    11: iload_3
    12: invokevirtual #13                 // Method java/io/PrintStream.println:
(I)V
    15: return
```

The program prints 10. The stack is [5, 2] just before `imul` and [10] just after.

5.9 The building blocks of the language

When the compiler reads a Java file, it first breaks it into small pieces and then builds larger structures from those pieces, just as letters form words, words form sentences, and sentences form paragraphs. This section starts with the smallest pieces, tokens, and reaches the concepts of expression, statement, and block.

5.9.1 Tokens

Definition 5.6

A **token** is the smallest piece of source code that is meaningful to the compiler. Java has five kinds of tokens: **keywords**, **identifiers**, **literals**, **operators**, and **separators**.

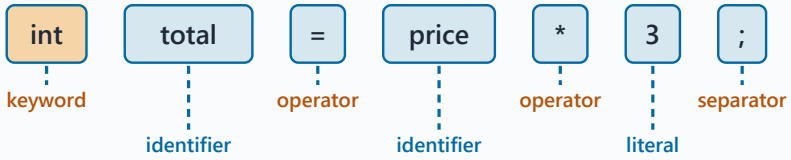


Figure 5.21: The seven tokens of the statement `int total = price * 3;`: `int` is a keyword, `total` and `price` are identifiers, `=` and `*` are operators, `3` is a literal, and `;` is a separator.

Let us get to know the five kinds with Figure 5.21:

- **Keywords** are words reserved for the language itself: `int`, `class`, `public`, `static`, `void`, `if`, and so on. Their meanings are fixed, and they cannot be used for any other purpose. `int class = 5;` cannot be written.
- **Identifiers** are names given by the programmer: variable, method, and class names. `total`, `price`, `Hello`, and `main` are identifiers. An identifier starts with a letter, `_`, or `$` and continues with letters and digits. It cannot contain a space and cannot start with a digit.
- **Literals** are values written directly into the code: `3`, `120`, `2.5`, `"Hello"`, `'A'`, `true`.
- **Operators** are the symbols that perform operations: `+`, `-`, `*`, `/`, `=`, `>`, and so on.
- **Separators** are the symbols that determine the structure: `;` `,` `.` `(` `)` `{` `}` `[` `]`.

Java is a **case-sensitive** language. `count` and `Count` are two different identifiers. `System` is right and `system` is wrong. There is no keyword `Int`; there is only `int`. Statements usually end with a semicolon; blocks are enclosed in curly braces.

5.9.2 Whitespace, indentation, and braces

The following two programs are identical. The first is written on a single line, the second is indented according to Java convention:

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

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

Both programs compile and turn into the same bytecode. The `main` instructions in the `javap -c` output of the one-line version are the same four as in the Hello program: `getstatic`, `ldc`, `invokevirtual`, `return`. To the compiler, spaces, tabs, and newlines only separate tokens. `x=1;` and `x = 1;` are the same. But `publicclass` counts as a single word and is not read as two tokens. (The two class files are not byte-for-byte identical, because a `.class` file also stores line numbers for error messages. The instructions, however, are the same.)

If whitespace does not matter to the compiler, why does indentation matter? Because code is written for humans as much as for the compiler. **Indentation** shows the reader how blocks are nested. In the Java convention the rules are these:

- Each new block starts 4 spaces further in than the previous one.
- The opening brace `{` stays at the end of the line.
- The closing brace `}` is on its own line and lines up with the line that opened it.

When these rules are followed, you can see at a glance which block a `}` closes. In code with broken indentation it is very hard to notice that a brace is missing.

There is one exception: spaces inside a string are part of the text. `"Hello "` and `"Hello"` are different strings; the first has six characters, the second five. `print("Hello ")` and `print("Hello")` give different output.

5.9.3 Expressions, statements, and blocks

Tokens come together to form expressions, and expressions form statements.

> **Definition 5.7**
>
> An **expression** is a piece of code that produces a value when evaluated: `2 + 3` → 5, `num > 0` → `true` or `false`. A **statement** is a unit of code that completes an action and usually ends with `;`: `total = 5;`, `System.out.println(total);`. A **block** groups the statements inside curly braces into a single structure.

The difference is clear from the question: "Is this piece a value or an action?" `price * qty` is a value (such as 360); by itself it does nothing. `total = price * qty;`, however, is an action: it computes the value and writes it to `total`. A statement can contain one or more expressions.

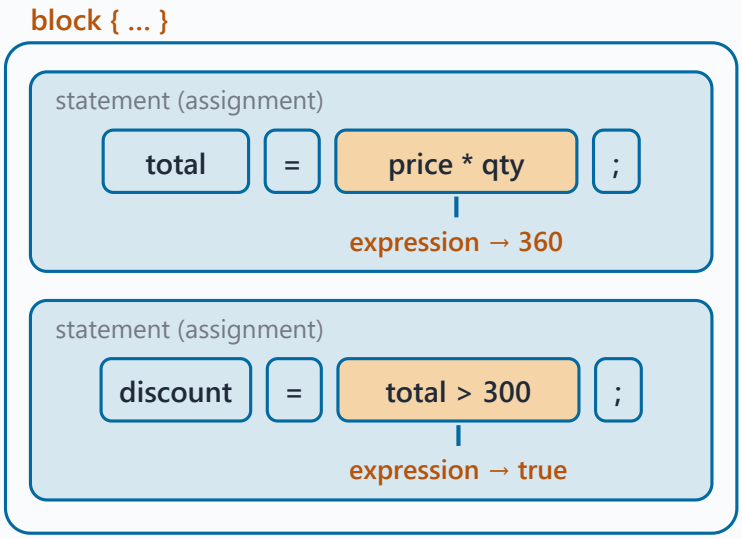


Figure 5.22: A block contains statements, and statements contain expressions. The expression of the first assignment statement in the block, `price * qty`, produces the value 360; the expression of the second, `total > 300`, produces the value `true`.

The following piece of code uses all four concepts together. When the piece is placed in the `main` method of a class and run, a single line is written to the console:

```
int price = 120;           // declaration
int qty = 3;               // declaration
int total = price * qty;   // expression + assignment
if (total > 300) {         // condition + block
    System.out.println("Discount");
}
```

```
Discount
```

Step	Line	What happens	price	qty	total	Console
1	1	an `int` box is set aside in memory for `price`, and 120 is written	120	?	?	
2	2	the `qty` box is created	120	3	?	
3	3	the expression on the right is evaluated: `price * qty` → 120 * 3 → 360	120	3	?	
4	3	the value produced is assigned to `total`	120	3	360	
5	4	the condition expression: `total > 300` → 360 > 300 → `true`; the block is entered	120	3	360	
6	5	the statement in the block runs	120	3	360	`Discount`

Step	Line	What happens	price	qty	total	Console
7	6	the block ends	120	3	360	`Discount`

Table 5.15: A step-by-step trace of the piece of code. "?" marks a variable that has not been given a value yet.

The first and second lines are **declaration statements**: each defines a variable and gives it its initial value. In the third line the expression is evaluated first, and then the assignment is made (Table 5.15, steps 3 and 4). The right-hand side of an equals sign is always computed first. The `if` statement on the fourth line consists of a condition expression and a block. Because the condition comes out `true`, the block runs. If `qty` were 2, `total` would be 240, the condition would be `false`, and the block would not run at all. A block makes several statements a single unit: if the condition is true, all the statements in the block run; if it is false, none of them does.

5.10 Comments

Code is written not only for the compiler but also for the people who will read it: the programmer returning to their own code months later, teammates, the instructor who grades the course. A **comment** is text added to the source code for the reader, which the compiler ignores.

5.10.1 Three comment styles

Java has three comment styles:

- A comment that starts with `//` lasts until the end of the line. It is used for short notes.
- A comment that starts with `/*` and ends with `*/` can span several lines.
- A comment that starts with `/**` and ends with `*/` is a **Javadoc** comment. The `javadoc` tool generates API documentation from these comments. They are usually written immediately above class and method declarations.

```
/** The Hello program (Javadoc). */
public class Hello {
    /* A comment spanning
       several lines */
    public static void main(String[] args) {
        // print the greeting
        System.out.println("Hello");
    }
}
```

```
> java Hello
Hello
> javap -c Hello
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #7                  // Field
java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #13                 // String Hello
       5: invokevirtual #15                 // Method java/io/PrintStream.println:
(Ljava/lang/String;)V
       8: return
}
```

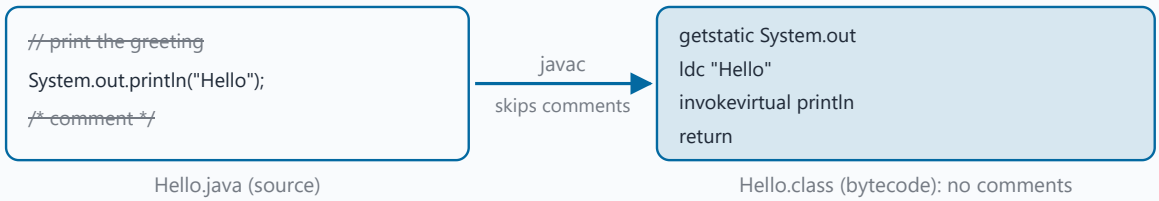


Figure 5.23: `javac` skips comments. The lines `// print the greeting` and `/* comment */` in the source do not enter the bytecode; the `.class` file contains only instructions.

The `javap` output of the commented program is the same as that of the uncommented Hello program (Figure 5.23). The compiler skips comments; they do not enter the bytecode and do not slow the program down.

A good comment, instead of repeating *what* the code does, says *why* it does it. The comment `i = i + 1; // increase i by one` is useless; anyone looking at the code already sees that. `total = total / 2; // the two exams have equal weight`, however, gives information that cannot be derived from the code.

5.10.2 Comment pitfalls

Comments look simple, but they hide a few pitfalls.

Nested multi-line comments. `/* … */` comments cannot be nested. A comment ends at the first `*/` it meets:

```
public class Nested {
  public static void main(String[] args) {
    /* outer /* inner */ rest */
    System.out.println("Hello");
  }
}
```

```
Nested.java:3: error: illegal start of expression
    /* outer /* inner */ rest */
                              ^
1 error
```

The compiler treats the part `/* outer /* inner */` as a comment; the second `/*` in the middle has no meaning because it is inside the comment. The comment ends at the `*/` after `inner`. What remains is `rest */`, and the compiler takes it for code. `rest` counts as an identifier and `*` as the multiplication operator. After the multiplication a value is expected, but a `/` comes, and no expression can start with `/`. The compiler says `illegal start of expression`; the `^` mark points at this last `/` character. To disable a section that contains `/* */`, it is safer to comment out the lines one by one with `//`.

Comment markers inside quotes. `//` or `/* */` inside a string is not a comment; it is part of the text. Let us trace which lines run in the following program:

```
public class CommentTest {
    public static void main(String[] args) {
        // System.out.println("1");
        System.out.println("2"); // System.out.println("3");
        /* System.out.println("4");
        System.out.println("5"); */
        System.out.println("6 // not a comment");
        System.out.println("7 /* not a comment */");
        System.out.println("a // b");
    }
}
```

```
2
6 // not a comment
7 /* not a comment */
a // b
```

The first line is a comment from start to end. On the second line `"2"` is printed; what comes after `//`, that is, the statement that would print 3, is a comment. `/* … */` comments out two lines (4 and 5) together. The `//` and `/* */` markers in the last three lines are inside quotes, so they are printed

as they are.

Disabling code temporarily. Putting `//` at the start of a line makes it not run without deleting it. This is a method often used when hunting for a bug. In IDEs, the Ctrl+/ shortcut turns the selected lines into comments in this way and reverses it.

Stale comments. The most insidious pitfall is a comment that contradicts the code:

```
double price = net * 1.20; // VAT 18%
```

The code compiles and works correctly: the VAT rate was updated and the code was written for 20%. But the comment states the old rate and misleads the reader. When a comment and the code contradict each other, the compiler runs the code; the code is always what is right. When the code is changed, the related comment must be updated too.

Example 5.4

What does the following program print?

```
public class Comments {
    public static void main(String[] args) {
        int x = 5; // x = 10;
        /* x = 20; */ x = x + 1;
        System.out.println("x = " + x); // x = 100
        System.out.println("// x = " + (x * 2));
    }
}
```

On line 3, `x` becomes 5; `// x = 10;` is a comment. On line 4, `/* x = 20; */` is a comment, but `x = x + 1;` on the same line is outside the comment and runs: `x` becomes 6. Line 5 prints `x = 6` ; the `// x = 100` at the end of the line is merely a wrong note. The `//` on line 6 is inside quotes, so it is part of the text; `x * 2` is computed first in the parentheses and gives 12:

```
x = 6
// x = 12
```

5.11 Errors

Every programmer makes mistakes. What sets a good programmer apart is being able to find and fix an error quickly. The first step is to recognize the kind of error. When an error shows up determines who catches it and how to look for it.

5.11.1 Kinds of errors

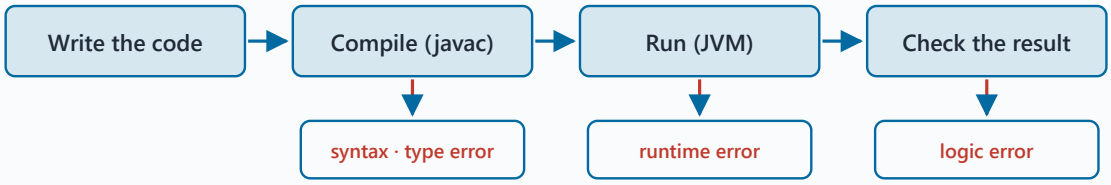


Figure 5.24: The stages at which errors are caught. Syntax and type errors are found at compile time (`javac`), runtime errors while the program runs (the JVM), and logic errors only when the result is checked.

Error	When?	Who catches it?	Example
Syntax	at compile time	`javac`	a missing semicolon
Type	at compile time	`javac`	assigning text to a number variable
Runtime	while the program runs	JVM	dividing an integer by zero
Logic	the program runs but the result is wrong	testing only	a wrong order of operations

Table 5.16: Kinds of errors.

A **syntax error** is a violation of the language's rules of writing: a missing semicolon, an unclosed parenthesis or quote. A **type error** is when the syntax is correct but a value is used where it does not fit its type. These two together are called **compile errors**. A program with a compile error does not turn into a `.class` file and does not run at all.

A **runtime error** is when a compiled program meets an operation it cannot perform while running. In Java, the JVM then throws an exception and the program stops unless special provision has been made. A **logic error** is the most insidious kind: the program compiles, runs without error, and produces a result, but the result is wrong. No tool gives an error message. A logic error can be caught only by someone who compares the result with the expected value.

The earlier errors are caught, the cheaper they are to fix. A compile error is found on the programmer's computer, within seconds. A logic error, however, may be noticed only after the program has reached users. That Java checks types at compile time is therefore an advantage: the programmer sees many mistakes before even running the program.

5.11.2 Four errors, four examples

The following four programs give one example of each kind of error. The real message on JDK 21 is given for each.

Syntax error. There is no semicolon at the end of line 3:

```
public class Bug1 {
    public static void main(String[] args) {
        System.out.println("Hello")
    }
}
```

```
Bug1.java:3: error: ';' expected
        System.out.println("Hello")
                                   ^
1 error
```

The compiler shows where the rule was broken with a line number: `Bug1.java:3`. `';' expected` means "a semicolon was expected." The `1 error` on the last line is the total number of errors. No `.class` is produced. The fix: add `;` at the end of the line.

Type error. A string is being put into a variable of type `int`:

```
public class Bug2 {
    public static void main(String[] args) {
        int num = "42";
    }
}
```

```
Bug2.java:3: error: incompatible types: String cannot be converted to int
        int num = "42";
                  ^
1 error
```

The syntax is flawless: type, name, equals sign, value, semicolon. But because `"42"` is inside double quotes, it is not a number but a two-character text. The message says "incompatible types: String cannot be converted to int." The `^` points at the incompatible value. The fix: remove the quotes and write `int num = 42;`.

Runtime error. The program compiles, but while running an integer is divided by zero:

```java
public class Bug3 {
    public static void main(String[] args) {
        int a = 10, b = 0;
        System.out.println(a / b);
    }
}
```

```
> javac Bug3.java
> java Bug3
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Bug3.main(Bug3.java:4)
```

`javac` stays silent: the syntax and the types are correct. The compiler does not check that the value of `b` will be 0. While the program runs, an integer division by zero is attempted on line 4. Faced with this operation, which is undefined in mathematics too, the JVM throws an `ArithmeticException`. The message says `/ by zero` and gives the location of the error: `Bug3.java`, line 4, the `main` method. The program stops there.

Logic error. The average of two grades is wanted:

```java
public class Bug4 {
    public static void main(String[] args) {
        int a = 70, b = 90;
        System.out.println(a + b / 2);
    }
}
```

```
> javac Bug4.java
> java Bug4
115
```

There is no error message, but the result is wrong. The average of 70 and 90 is 80, and the program prints 115. The reason is operator precedence: `/` is performed before `+`. The expression `a + b / 2` is evaluated as `a + (b / 2)`: 90 / 2 = 45, 70 + 45 = 115. The fix: write `(a + b) / 2`; this expression gives (70 + 90) / 2 = 160 / 2 = 80.

> **Common mistake**
>
> Concluding "the program compiled and ran, so it is correct." Compilation shows only that there are no syntax or type errors. A program with a logic error also compiles and runs without trouble. A program can be trusted to be correct only by comparing its output with the

expected results for known inputs.

5.11.3 Common compile and launch errors

Most of the compile errors beginners meet fit a few patterns. The table below gives them with the real messages of JDK 21.

Error	Real message	Fix
Missing semicolon	`Hello.java:3: error: ';' expected`	add `;` at the end of the previous statement
Wrong method name (`System.out.printn`)	`error: cannot find symbol` `symbol: method printn(String)` `location: variable out of type PrintStream`	make the name `println`
Lowercase letter (`system.out`)	`error: package system does not exist`	write `System` with a capital S
Class name and file name do not match	`error: class Greeter is public, should be declared in a file named Greeter.java`	make the file name the same as the public class name
Wrong way of running (`java Greeter`)	`Error: Could not find or load main class Greeter`	there is no `.class` file, or the name or location is wrong; compile first, and do not write the extension

Table 5.17: Common errors and their messages. The first four are compile errors, the last is a launch error.

The full form of the messages shows more clearly where the problem is. When a method name is misspelled:

```
Hello.java:3: error: cannot find symbol
        System.out.printn("Hello");
                   ^
  symbol:   method printn(String)
  location: variable out of type PrintStream
1 error
```

`cannot find symbol` says that the compiler does not recognize a name (a variable, method, or class name). The `symbol:` line below gives which name was not found, and the `location:` line where it was looked for. According to this message, a method `printn` that takes a `String` was looked for in the variable `out` of type `PrintStream` and was not found. The `^` mark points at the dot after `out`; the method was being attached there.

When `System` is written in lowercase, the message is surprising at first sight:

```
Hello.java:3: error: package system does not exist
        system.out.println("Hello");
              ^
1 error
```

The compiler knows no class named `system`. Seeing the dots in `system.out.println`, it thinks `system` might be a package name and cannot find such a package either. That is why the message says "package"; the real problem is letter case. The same happens for a misspelled `System`: for `Sytem.out.println` the message is `package Sytem does not exist`.

If the name of the public class differs from the file name, the compiler reports the error on the class declaration line:

```
public class Greeter {
    public static void main(String[] args) {
        System.out.println("x");
    }
}
```

```
> javac Hello.java
Hello.java:1: error: class Greeter is public, should be declared in a file named
Greeter.java
public class Greeter {
       ^
1 error
> java Greeter
Error: Could not find or load main class Greeter
Caused by: java.lang.ClassNotFoundException: Greeter
```

Because compilation failed, `Greeter.class` was not produced either; the following `java Greeter` command cannot find the class. Letter case counts too. If the class `Hello` is saved in the file `hello.java`, the compiler distinguishes case even though Windows file names do not:

```
hello.java:1: error: class Hello is public, should be declared in a file named Hello.java
public class Hello {
       ^
1 error
```

There are two more common compile errors. An unclosed quote makes everything up to the end of the line part of the string:

```
Quote.java:3: error: unclosed string literal
        System.out.println("Hello);
                           ^
1 error
```

In the "unclosed string literal" message, the `^` points at the quote where the string began. A missing closing brace, on the other hand, is noticed at the end of the file. In the message below, the last `}` that closes the class has been deleted:

```
Hello.java:4: error: reached end of file while parsing
    }
     ^
1 error
```

The message "reached end of file while parsing" says that the compiler was still expecting a block to close when the file ended. The real location of the error is not the end of the file but the place where the `}` is missing. In correctly indented code this place is easy to find.

5.11.4 Reading an error message

Error messages look frightening at first sight, but they are always written in the same layout. Learning the layout once makes every message easier to read.

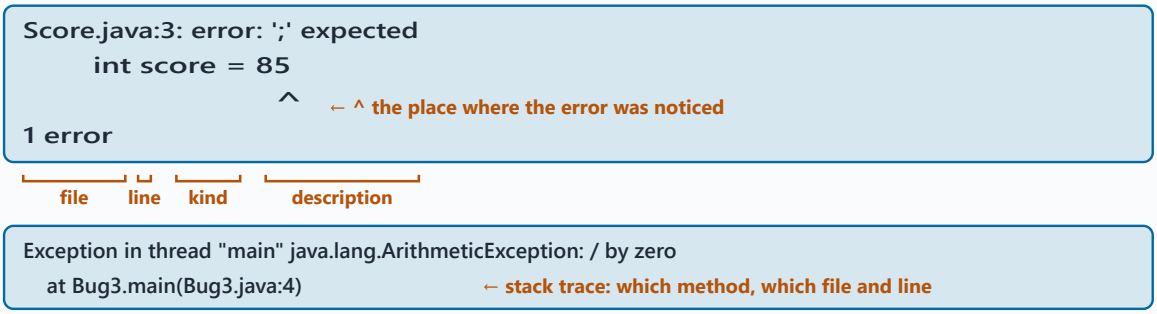


Figure 5.25: The parts of an error message. Top, a `javac` message: file, line, kind, description; below it the offending source line and a `^` that points at the place where the error was noticed. Bottom, a runtime message: the type and detail of the exception, followed by the stack trace, which says in which method and at which line of which file it occurred.

The first line of a `javac` message has the form `file:line: error: description` (Figure 5.25). In the message `Score.java:3: error: ';' expected`, the file is `Score.java`, the line is 3, the kind is `error`, and the description is `';' expected`. The second line shows the offending source line verbatim. The `^` mark on the third line points at the column where the compiler noticed the problem. At the very end the total number of errors is written.

The `^` does not always point at the error itself but at the place where the compiler *noticed* it. A missing semicolon is noticed where the statement ends, that is, at the end of the line. A missing `}` is not noticed until the end of the file and is reported as `reached end of file while parsing`. So if the error cannot be found at the place shown, look at the previous line or at the blocks above.

Runtime messages have a different layout. The first line begins with `Exception in thread "main"` and gives the type and detail of the exception: `java.lang.ArithmeticException: / by zero`. The lines below it that begin with `at` are the **stack trace**: they list which methods were on the call stack at the moment of the error, starting from the innermost method. `at Bug3.main(Bug3.java:4)` says the error occurred in the `main` method of the class `Bug3`, at line 4 of the file `Bug3.java`. In the first programs the stack trace is a single line, because only `main` runs. As methods call one another, the trace gets longer.

5.11.5 Fixing errors in order

A program can have more than one error. The compiler does not always report them all at once. The following program has two errors: the semicolon is missing on line 3, and the method name is wrong on line 4.

```
public class Score {
    public static void main(String[] args) {
        int score = 85
        System.out.printn("Score: " + score);
    }
}
```

The first compilation reports only one error:

```
Score.java:3: error: ';' expected
        int score = 85
                    ^
1 error
```

The mistake on line 4 is not visible yet. The reason is that the compiler works in stages. It parses the syntax first. As long as the syntax is broken, it does not go on to semantic checking, that is, checking names and types. The error is fixed, line 3 becomes `int score = 85;`, and the file is saved. This time the second compilation reports a different error:

```
Score.java:4: error: cannot find symbol
        System.out.printn("Score: " + score);
                   ^
  symbol:   method printn(String)
  location: variable out of type PrintStream
1 error
```

The syntax is now correct; semantic checking runs and cannot find a method named `printn`. The `symbol:` line states the name that was not found. The method name is corrected to `println`. The third compilation succeeds silently, `Score.class` is created, and the program runs:

```
> javac Score.java
> java Score
Score: 85
```

Compilation	line 3	line 4	Error reported	`Score.class`
1	`int score = 85` ✗	`printn` ✗	`Score.java:3: error: ';' expected`	none
2	`int score = 85;` ✓	`printn` ✗	`Score.java:4: error: cannot find symbol`	none
3	`int score = 85;` ✓	`println` ✓	none	created

Table 5.18: Fixing the errors in order.

The lesson of this example is that after fixing one error, the number of errors may not decrease, and may even increase. This does not mean the fix was wrong; the compiler has merely moved on to the next stage. Conversely, errors in the same stage are reported together. The two type errors in the following program appear in a single compilation:

```
Two.java:3: error: incompatible types: String cannot be converted to int
        int a = "5";
                ^
Two.java:4: error: incompatible types: String cannot be converted to int
        int b = "6";
                ^
2 errors
```

A syntax error, however, can also confuse the lines that follow it, and errors that do not really exist may be reported. For this reason, always start with the **first error**. Fix the first error, recompile the file, and read the new messages.

5.12 Before coding

Knowing how to read error messages is important, but the cheapest error is the one never made. Most logic errors can be prevented by a little preparation on paper before the code is written.

5.12.1 A program is a recipe

A recipe lists the ingredients, gives the steps in order, and says what will be obtained at the end. A program has the same structure. The inputs correspond to the ingredients, the statements to the steps of the recipe, and the output to the finished dish. In a good recipe it is clear what each step requires and what it produces: "After the dough has risen (precondition), it goes into the oven and bakes for 30 minutes (step); the bread browns (result)."

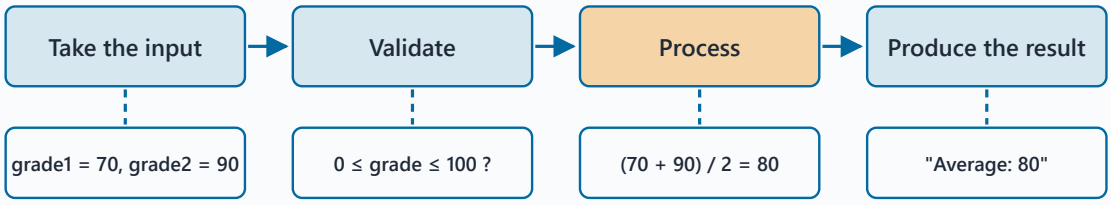


Figure 5.26: The recipe for the average program: take the input (grade1 = 70, grade2 = 90), validate it (is 0 ≤ grade ≤ 100?), process it ((70 + 90) / 2 = 80), produce the result ("Average: 80").

Figure 5.26 gives the recipe of a program that finds the average of two grades in four steps. The precondition and the value produced by each step are clear:

1. **Take the input.** There is no precondition. It produces the values `grade1` and `grade2`.
2. **Validate.** Precondition: two grades have been read. Each grade is checked to be between 0 and 100. If there is an invalid grade, the recipe stops here and gives an error message.
3. **Process.** Precondition: both grades are valid. The average is computed as `(grade1 + grade2) / 2`.
4. **Produce the result.** Precondition: the average has been computed. A line such as "Average: 80" is printed.

The recipe should be tested by hand before coding. When the steps are followed for 70 and 90, does 80 come out? 160 / 2 = 80; yes. If 101 is entered, which step stops it? The second: because 101 > 100, the validation fails. Testing by hand catches logic errors before any code is written. In the order-of-operations example (a + b / 2), the error could have been seen at the recipe stage, because the recipe clearly says "first add, then divide."

Large recipes are divided into sub-recipes. In a cake recipe, "prepare the cream" can be a recipe of its own. In programs too, large jobs are divided into subtasks; in Java each subtask is written as a method. For the first programs a single `main` method is enough, but the recipe way of thinking should become a habit from the start.

5.12.2 Sample input and expected output

There are four jobs to do before starting to code:

1. Write down sample inputs and the **expected output** for each.
2. Break the solution into steps in natural language or pseudocode.
3. Determine the **boundary cases** and what to do on invalid input.
4. Only then translate the steps into Java constructs.

The product of the first step is a small table. For the average program, a table like this can be prepared:

Sample input	Expected output	Why?
70, 90	80	typical case
0, 0	0	lower bound
101, 50	error message	invalid input

Table 5.19: Sample inputs and expected outputs for the average program.

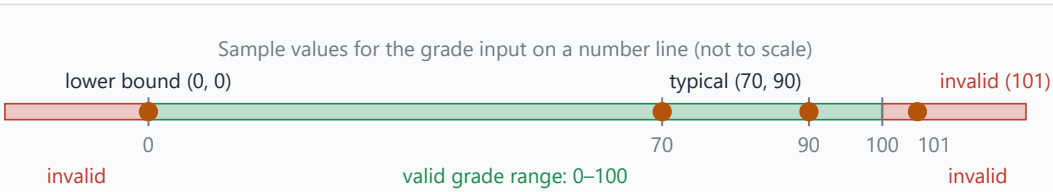


Figure 5.27: Sample values for the grade input on a number line (not to scale). The valid range is 0–100. 0 is the lower bound, 70 and 90 are typical values, and 101 is an invalid value.

Each row of the table tests something different (Figure 5.27). The **typical case** shows whether the program works correctly in everyday use. The **boundary case** is the values at the ends of the valid range; this is where programs most often go wrong. A small slip such as writing `<=` instead of `<` shows up only at the boundary. The **invalid input** shows what the program does when faced with an unexpected value: it must not crash; it must give an understandable message.

The table has one more benefit: once the program is written, each row becomes a test. The program is run with these inputs and its output is compared with the expected column. That is why choosing different kinds of examples matters. A program tested only with typical inputs can hide a mistake

such as the fractional part being discarded in the quotient of two integers. For example, for 75 and 80 the correct average is 77.5, but the quotient of two `int`s gives 77. This mistake does not show up until an input with an odd sum is tried.

5.13 Summary

- A **program** is a whole made of commands that a computer executes to perform a task; it takes input, processes it, keeps state, and produces output. A programming language is studied from the aspects of **syntax**, **semantics**, **type system**, and **execution model**.
- **Source code** is written in a plain-text `.java` file. The character encoding (UTF-8), the line endings, and the file name must be consistent. The editor, the IDE, `javac`, `java`, `javadoc`, and version control are the tools on the program's journey.
- Java was born at Sun Microsystems in 1995; it has been developed by Oracle since 2010, and its source code is in OpenJDK. Its design goal is the principle of **write once, run anywhere**. A new release comes out every six months; 8, 11, 17, 21, and 25 are **LTS** releases.
- **JDK** = JRE + development tools; **JRE** = JVM + class library. The **JVM** executes bytecode; it contains the class loader, the verifier, the interpreter, the JIT compiler, and the garbage collector.
- All Java code is inside a class. The name of the public class is the same as the file name. The JVM starts running at the method with the signature `public static void main(String[] args)`. A `main` with a wrong signature compiles but cannot be launched. Java is case-sensitive.
- **Command-line arguments** reach `main` through the `args` array; indices start at 0, and `args.length` is the number of arguments. Accessing an argument that does not exist throws `ArrayIndexOutOfBoundsException` at run time.
- `println` adds a newline after writing; `print` does not. **Escape sequences** (`\n`, `\t`, `\"`, `\\`, `\'`, `\u`) put special characters into a string. In Windows paths, a single backslash either gives a compile error or silently produces wrong output.
- `+` adds between two numbers and concatenates if one of the operands is a string. Operations are performed left to right; `*` and `/` come first: `"Score: " + 40 + 45` → `Score: 4045`, `"Score: " + (40 + 45)` → `Score: 85`.
- `javac Hello.java` compiles and produces `Hello.class`; `java Hello` runs it. When the source changes, it must be recompiled, because the JVM runs only the `.class` file. If there is a compile error, no `.class` is produced.
- **Bytecode** is processor-independent and is stored in files that begin with `CA FE BA BE`. The JVM is a **stack machine**: `iload` pushes values onto the **operand stack**, `iadd` adds the values on the stack, and `istore` writes the result to a local variable. `javap -c` shows bytecode in readable form.
- The compiler splits code into **tokens**: keyword, identifier, literal, operator, separator. Whitespace only separates tokens; indentation is for the reader. An **expression** produces a value, a **statement** completes an action, and a **block** groups statements.

- Comments are written with `//`, `/* */`, and `/** */` and do not enter the bytecode. `/* */` cannot be nested; comment markers inside quotes are part of the text.
- **Syntax** and **type errors** are found at compile time, **runtime errors** while running, and **logic errors** only when the result is checked. A `javac` message has the form `file:line: error: description`; the `^` shows where the error was noticed. The **stack trace** in a runtime message gives the method and the line of the error. Errors are always fixed starting from the first error.
- Before coding, sample inputs, expected outputs, boundary cases, and error behavior are determined; the recipe is tested by hand.

5.14 Key terms

The key terms of this chapter are listed below.

program syntax semantics type system execution model source code source file character encoding compiler compilation bytecode Java Virtual Machine (JVM) Java Runtime Environment (JRE) Java Development Kit (JDK) class library class loader bytecode verifier just-in-time (JIT) compilation garbage collection long-term support (LTS) integrated development environment (IDE) Javadoc class declaration main method case-sensitive command-line argument console newline escape sequence tab stop Unicode escape string concatenation package environment variable stack machine operand stack token keyword identifier literal separator expression statement declaration statement comment syntax error type error compile error runtime error logic error exception stack trace error message boundary case

---

5.15 Test Yourself

1. A computer has only a JRE installed and no JDK. Can `Hello.java` be compiled on this computer? Can a `Hello.class` compiled on another computer be run?

   Answer

   It cannot be compiled: `javac` is a development tool and exists only in the JDK. It can be run: the JRE contains the JVM and the class library, which is enough for `java Hello`. With one condition: the `.class` file must not have been compiled with a JDK newer than the JRE's version. Otherwise you get an `UnsupportedClassVersionError`.

2. What is the output of the following program, and how many lines does it produce?

```
public class TY2 {
    public static void main(String[] args) {
        System.out.print("1");
        System.out.print("2\n3");
        System.out.println("4");
        System.out.print("\"5\"");
        System.out.println();
    }
}
```

Answer

```
12
34
"5"
```

Three lines are produced. "1" and "2" are written on the same line; the `\n` in `"2\n3"` ends the line and "3" starts on the line below. `println("4")` writes "4" on the same line and ends it. Because `\"` writes a double quote, the third line is `"5"`; the final `println()` ends this line.

3. What is the output of the following program? In each line, which of the `+` signs is addition and which is concatenation?

```
public class TY3 {
    public static void main(String[] args) {
        System.out.println("Result: " + 10 / 4 + 1);
        System.out.println(10 / 4 + 1 + " items");
        System.out.println("Result: " + (10 / 4 + 1));
    }
}
```

Answer

```
Result: 21
3 items
Result: 3
```

Line 1: `/` is performed first, 10 / 4 = 2 (integer division). Then `"Result: " + 2` is concatenation and gives `"Result: 2"`; `+ 1` is also concatenation: `Result: 21`. Line 2: 10 / 4 = 2, and `2 + 1` is addition because it is between two numbers and gives 3; `3 + " items"` is concatenation. Line 3: the parentheses are computed first, 2 + 1 = 3; then concatenation is done.

4. The program `B` is below. What happens when it is run with the commands `java B` and `java B Ece`? What kind of error is it?

```java
public class B {
    public static void main(String[] args) {
        System.out.println("First argument: " + args[0]);
    }
}
```

Answer

```
> java B
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 0 out
of bounds for length 0
        at B.main(B.java:3)
> java B Ece
First argument: Ece
```

When no argument is given, `args` is an empty array (length 0) and `args[0]` does not exist. The JVM throws `ArrayIndexOutOfBoundsException` on line 3. Because the program compiled and the error occurred while running, depending on the input, it is a runtime error. With a single argument the program runs without trouble.

5. What kind of error is each of the following: syntax, type, runtime, or logic? (a) `Public class D {` (b) `int x = 3.5;` (c) Writing `2 * a + b` for the perimeter of a rectangle (d) `int a = 10, b =`

`0;` then `a / b`.

Answer

(a) A syntax error: `Public` with a capital letter is not a keyword, and the compiler expects a class declaration at the start of the line. The message: `class, interface, enum, or record expected`. (b) A type error: 3.5 is a fractional number and does not fit into `int`. The message: `incompatible types: possible lossy conversion from double to int`. (c) A logic error: the code compiles and runs, but the correct form is `2 * (a + b)`. (d) A runtime error: `ArithmeticException: / by zero`.

6. Does the statement `System.out.println("C:\temp\new");` compile? If it does, what does it print? How is the correct path written?

Answer

It compiles, because `\t` and `\n` are valid escape sequences. But the output is wrong: `\t` becomes a tab and `\n` a newline. The screen shows "C:", a tab's worth of space, and "emp", and "ew" on the next line:

```
C:      emp
ew
```

The correct way to write it is `"C:\\temp\\new"` or `"C:/temp/new"`.

7. For the `main` method of the following program, what are the first eight instructions in the `javap -c` output? Why does the order of the `iload` instructions matter?

```java
public class Subtract {
    public static void main(String[] args) {
        int a = 2;
        int b = 5;
        int c = b - a;
        System.out.println(c);
    }
}
```

Answer

```
public static void main(java.lang.String[]);
  Code:
     0: iconst_2
     1: istore_1
     2: iconst_5
     3: istore_2
     4: iload_2
     5: iload_1
     6: isub
     7: istore_3
          8:  getstatic          #7                                  //  Field
java/lang/System.out:Ljava/io/PrintStream;
    11: iload_3
    12: invokevirtual #13                 // Method java/io/PrintStream.println:
(I)V
    15: return
```

`a` is in slot 1 and `b` in slot 2. For `b - a`, first `b` (`iload_2`) and then `a` (`iload_1`) are pushed onto the stack. `isub` subtracts the top value from the value below it: 5 − 2 = 3. In addition the order does not change the result, but in subtraction it does; the loading order follows the order of the operands in the source code. The program prints 3.

8. `Hello.java` has been compiled and a `Hello.class` that prints `Hello` has been produced. Then in the source `"Hello"` is changed to `"Hi"` and saved, and without recompiling, the commands `java Hello`, `java Hello.java`, `java Hello` are run in turn. What does each command print?

Answer

```
> java Hello
Hello
> java Hello.java
Hi
> java Hello
Hello
```

`java Hello` runs the old bytecode in the `.class` file and prints `Hello`. `java Hello.java`, however, compiles the source file in memory on the spot and runs it, printing `Hi`; it does not touch the `.class` file on disk. So the last command again prints `Hello`. The `.class` is renewed only by `javac`.

9. The following program is compiled. How many compilations are needed for the program to

compile without errors? What message is received in each compilation?

```
public class Two {
    public static void main(String[] args) {
        System.out.println("A")
        system.out.println("B");
    }
}
```

Answer

Three compilations are needed. The first compilation reports only the syntax error; the lowercase mistake on line 4 is not visible yet:

```
Iki.java:3: error: ';' expected
        System.out.println("A")
                                ^
1 error
```

When `;` is added to line 3, the second compilation moves on to semantic checking and reports line 4:

```
Iki.java:4: error: package system does not exist
        system.out.println("B");
              ^
1 error
```

When `system` is changed to `System`, the third compilation succeeds. There were two errors, but the compiler showed only one each time.

10. A program is to be written that takes the side lengths of a rectangle and prints its area. Which sample inputs and expected outputs should be written in a table before coding?

Answer

A sample table: typical case, 3 and 4 → 12. A square, 5 and 5 → 25 (it must work when the two sides are equal too). A boundary case, 0 and 7 → 0 or the message "a side cannot be 0"; which of the two is wanted must be decided beforehand. Invalid input, −2 and 4 → an error message (a length cannot be negative). These last two rows bring out, before coding, decisions that were left missing in the problem definition.

5.16 Further reading

- Oracle, *The Java Tutorials*, "Getting Started" and "A Closer Look at the 'Hello World!' Application": explain every part of the first program, the `main` signature, and the compile-and-run steps from the official source, with short examples.
- R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), 1.1 Your First Program: covers the edit-compile-run cycle, command-line arguments, and common error messages; the book's website (introcs.cs.princeton.edu) has a chapter summary and exercises.
- A. B. Downey, C. Mayfield, *Think Java* (2nd ed., O'Reilly, 2019), Chapter 1: explains programs, compilation, kinds of errors, and escape sequences in plain language for beginners; the book has a free online version.
- Oracle, *The Java Language Specification, Java SE 21 Edition*, Chapter 3 "Lexical Structure": gives the precise definition of tokens, identifiers, comments, and escape sequences. Chapters 2 and 6 of *The Java Virtual Machine Specification* explain the operand stack and instructions such as `iload` and `iadd` in detail.

5.17 References

1. P. Deitel, H. Deitel, *Java: How to Program, Early Objects*, 11th ed., Pearson, 2018.
2. Y. D. Liang, *Introduction to Java Programming and Data Structures*, 13th ed., Pearson, 2024.
3. Oracle, *The Java Tutorials*: A Closer Look at the "Hello World!" Application.
4. Oracle, *The Java Language Specification, Java SE 21 Edition*: Chapter 3. Lexical Structure.
5. Oracle, *The Java Virtual Machine Specification, Java SE 21 Edition*: online version.
6. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 1.1 Your First Program.
7. A. B. Downey, C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019 (free online version).
8. OpenJDK, JEP 330: Launch Single-File Source-Code Programs (Java 11); JEP 400: UTF-8 by Default (Java 18); JEP 512: Compact Source Files and Instance Main Methods (Java 25).

PART III

Programming in Java

Chapters 6–12

Variables, Data Types and Operators

Every program works with data: a student's age, the price of a product, the average of an exam. A program keeps this data in named boxes in memory, that is, in variables. Every box has a type, and the type determines which values fit in the box. This chapter describes how to declare variables and give them values in Java, the eight primitive types and their ranges, the overflow that occurs when a limit is exceeded, and why decimal numbers cannot be stored exactly. Next come conversions between types, integer division, arithmetic, comparison and logical operators, operator precedence, and short-circuit evaluation. The last sections cover reading data from the keyboard with Scanner, producing formatted output with printf, and small programs that combine these tools. All Java code in the chapter was compiled and run with JDK 21; the outputs shown are the programs' actual outputs.

LEARNING OUTCOMES

Variables
Declare a variable, give it an initial value, and update its value.

Types
Recognize the primitive types and their ranges; choose a suitable type according to the risk of overflow and floating-point error.

Expressions
Use assignment, type conversion, and operators correctly; compute the value of an expression step by step with the precedence and type rules.

Input/Output
Read input from the keyboard with Scanner and produce formatted output with printf.

6.1 Variables

Someone who uses a calculator to find the average of the scores 70, 85, and 90 first adds the three numbers, keeps the intermediate result on the display, and then divides by 3. It is also possible to write the intermediate result on a piece of paper. A computer program has the same need: it must

store the numbers it reads, the intermediate results, and the final answer somewhere. That place is memory. The program gives a name to every place it sets aside in memory and reaches that place by its name.

6.1.1 What is a variable?

Definition 6.1

A **variable** is a named box in memory set aside for a value of a certain type. A variable has three properties: its **name**, its **type**, and its current **value**. The name and type stay fixed throughout the program; the value can change.

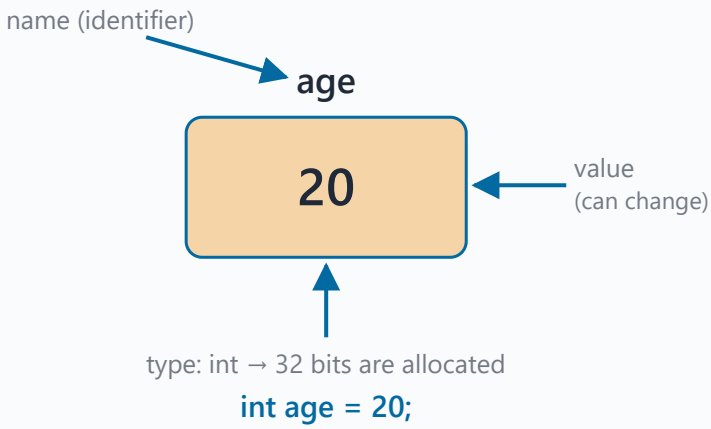


Figure 6.1: What the line `int age = 20;` corresponds to in memory. The name of the box is `age`, its type is `int`, and its value is 20. Space of 32 bits is allocated for the int type.

Figure 6.1 shows these three properties together. The name `age` above the box is the name the programmer has given to this box; in Java it is called an **identifier**. The value 20 inside the box can change while the program runs: when a new year begins, the age becomes 21. The `int` below the box is the type. A type determines two things. The first is the size of the box: for int, 32 bits, that is 4 bytes, are allocated. The second is the values that can be put in the box: int is an integer type, so 20 can be put in this box but 20.5 or “twenty” cannot.

The name “variable” comes from mathematics, but its meaning is slightly different. In mathematics, *x = x + 1* is an equation with no solution. In programming, this line is a command: “read the value in box x, add 1, and write the result back into box x.” A variable is a box that can hold different values over time; the equals sign is not a claim of equality but a write operation.

6.1.2 Declaration and initial value

In Java, a variable must be declared before it is used. A declaration tells the compiler, “I want a box with this name and this type.”

Definition 6.2

A **declaration** is a statement that specifies the type and name of a variable: `int age;`. **Initialization** specifies the variable's starting value. The two can also be written in a single statement: `double avg = 82.5;`.

The four lines in the slides show different forms of declaration:

```
int age;          // declaration
age = 20;        // assignment
double avg = 82.5; // both
final int MAX = 100; // constant
```

The first line is only a declaration: an int box named `age` is set aside, but no value has been put into it yet. The second line is an **assignment**: the value 20 is written into the `age` box. The third line combines the declaration with the initial value; the `avg` box is set aside and receives the value 82.5 at the same time. The `final` keyword in the fourth line turns the box into a **constant**: `MAX` receives the value 100 once and keeps it throughout the program.

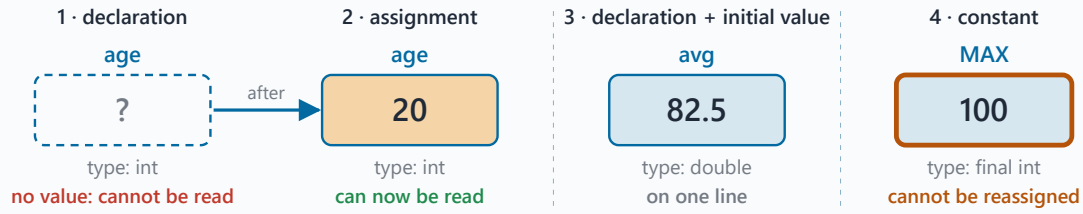


Figure 6.2: The effect of the four lines in memory. After the declaration, the `age` box has no value and cannot be read; after the assignment it is 20. `avg` is declared and given a value in one line. `MAX`, declared with `final`, cannot be reassigned.

Java calls variables declared inside a method **local variables**. There is a strict rule for local variables: a variable must definitely have been given a value before it is read. The compiler checks this rule by reading the code, without running the program. The following two-line fragment therefore does not compile:

```
int age;
System.out.println(age);
```

```
error: variable age might not have been initialized
System.out.println(age);
                   ^
1 error
```

The message says “the variable age might not have been given a value”. The compiler sees that the `age` box is empty and does not produce the program at all. This rule is a safety measure. Without it, the program could read whatever value happens to be in the box at that moment and produce meaningless results. In some older languages (C, for example) this really happens.

A similar check exists for `final` variables. Once a value has been given, a second assignment is a compile error:

```
final int MAX = 100;
MAX = 200;
```

```
error: cannot assign a value to final variable MAX
MAX = 200;
^
1 error
```

Constants are used for values that occur in several places in a program and must not change: the highest score, the VAT rate, the size of an array. Defining a constant instead of writing the number separately everywhere has two benefits. The code becomes readable; the expression `score <= MAX` says more than `score <= 100`. Also, if the value changes, only one line has to be corrected.

Common mistake

Assuming that a declared variable automatically gets the value 0. In Java, local variables do not get a value on their own; writing `total += 5;` after the line `int total;` gives a “might not have been initialized” compile error. A variable that will hold a running total is explicitly started at zero: `int total = 0;`.

6.1.3 Variables in memory, step by step

The best way to see how variables receive and change values is to trace a program line by line. The example program from the slides is:

```
int age;                  // declaration
age = 20;                 // first assignment
age = age + 1;            // update
double avg = 82.5;        // declare + initialize
avg += 5;                 // compound assignment
final int MAX = 100;      // constant
System.out.println(age);
System.out.println(avg);
```

The program produces two lines of output:

```
21
87.5
```

Table 6.1 gives the state of the boxes after each line has run. A question mark shows that the box has been set aside but has not yet received a value; a dash shows that the box has not been set aside at all yet.

Line	Statement	age	avg	MAX	Output
1	`int age;`	?	–	–	
2	`age = 20;`	20	–	–	
3	`age = age + 1;`	21	–	–	
4	`double avg = 82.5;`	21	82.5	–	
5	`avg += 5;`	21	87.5	–	
6	`final int MAX = 100;`	21	87.5	100	
7	`System.out.println(age);`	21	87.5	100	21
8	`System.out.println(avg);`	21	87.5	100	87.5

Table 6.1: Trace table of the memory example. Each row shows the state after the statement on that line has run.

The third line must be read carefully. The statement `age = age + 1;` runs in two stages. First the expression to the right of the equals sign is evaluated: the 20 in the `age` box is read and 1 is added, giving 21. At this stage the box has not changed yet. In the second stage the value 21 is written into the box on the left; the old 20 is erased. A box holds only one value at a time, and the new value overwrites the old one.

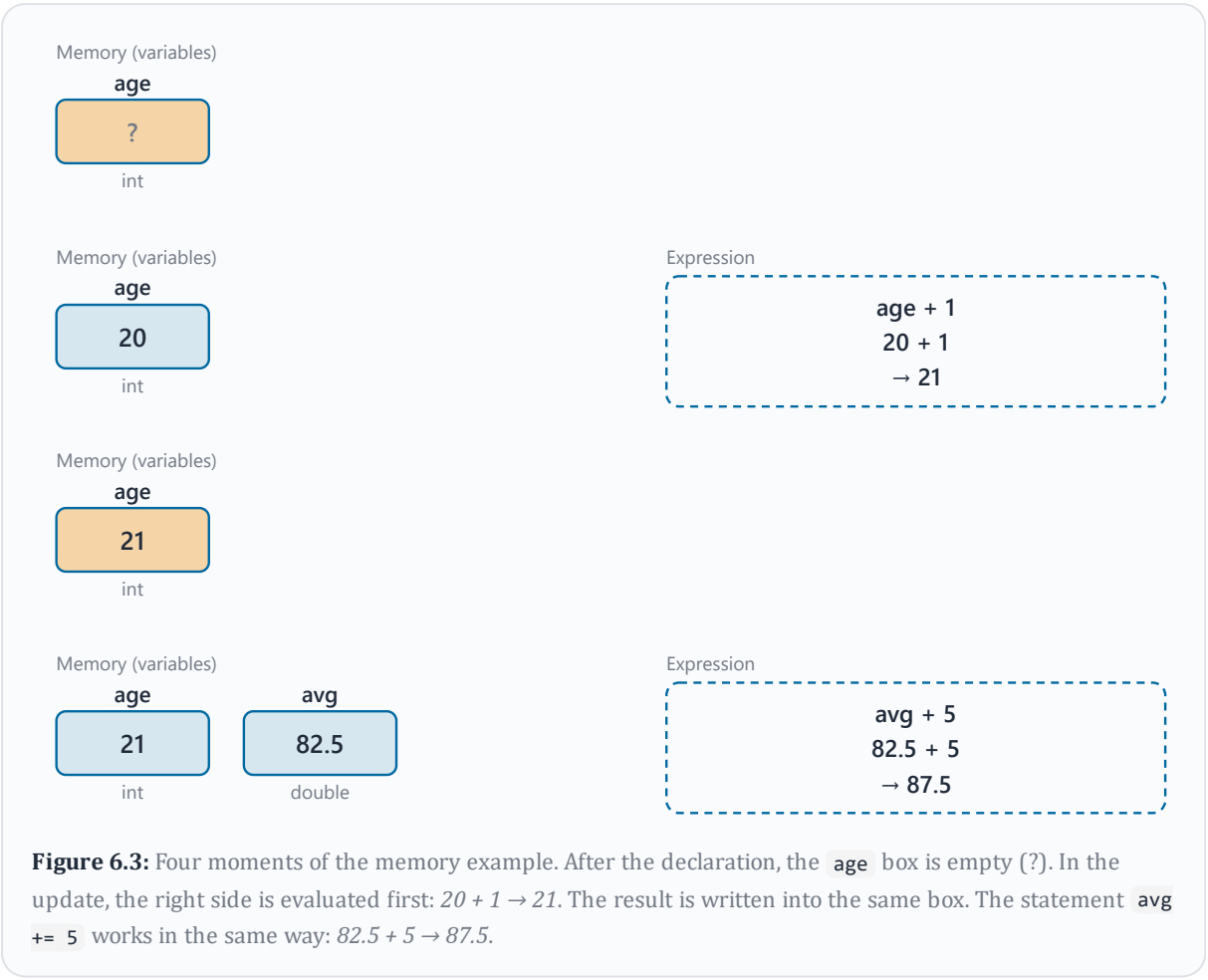


Figure 6.3: Four moments of the memory example. After the declaration, the `age` box is empty (?). In the update, the right side is evaluated first: *20 + 1 → 21*. The result is written into the same box. The statement `avg += 5` works in the same way: *82.5 + 5 → 87.5*.

The seventh and eighth lines only read the boxes. Reading copies a value out of the box but does not change the box; after the line `System.out.println(age)`, `age` is still 21. In the final state there are three boxes in memory and two lines of output on the screen. If `MAX = 200;` were written after the sixth line, the program would not compile; the value of a final variable is given once.

6.1.4 Assignment and updating

The assignment operator `=` is Java's most frequently used operator and always works in the same order: first the right side is evaluated, then the value is written into the variable on the left. On the left side there is always a single variable. Lines such as `5 = x;` or `x + 1 = 7;` are meaningless, because 5 and `x + 1` are not boxes.

The assigned value must be compatible with the variable's type. The string `"Ada"` cannot be assigned to an int box; the compiler reports this as a type mismatch error.

In programs it is very common to change a variable's value by using its own old value: adding a new price to a total, increasing a counter by one, subtracting an amount from a balance. Java has a short notation for this.

Definition 6.3

A **compound assignment** is an operator that combines an arithmetic operation with an assignment: `+=`, `-=`, `*=`, `/=`, `%=`. The statement `total += price` means `total = total + price`.

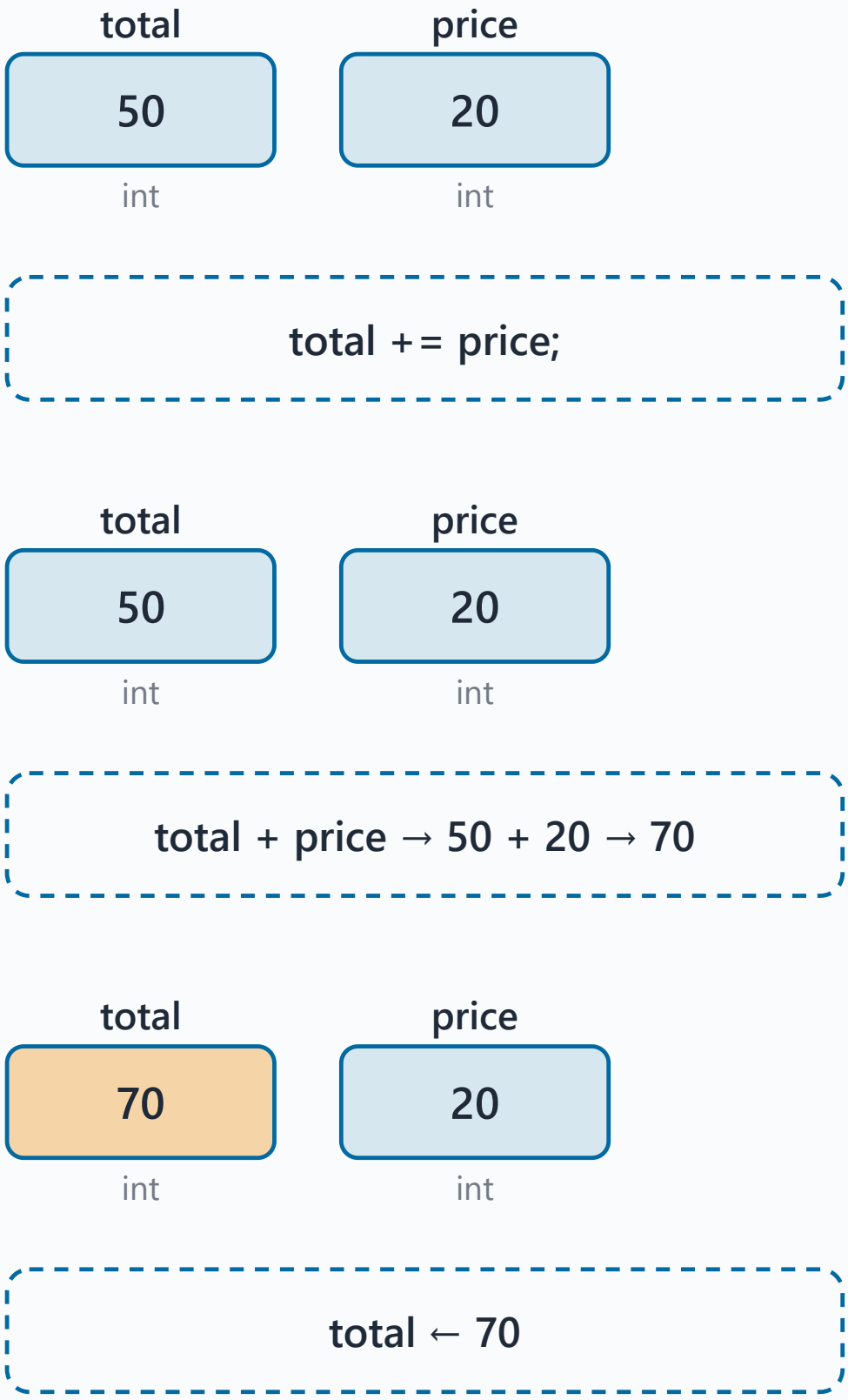


Figure 6.4: The three steps of the statement `total += price;`. Initially total = 50 and price = 20. First the two boxes are read and *50 + 20 → 70* is computed. The result is written into the box on the left: total = 70. price

does not change.

Figure 6.4 shows the steps of the operation. The variable on the right (`price`) is only read; the variable on the left (`total`) is both read and written. Compound assignment has one more subtlety: the right side is evaluated completely before the operation. The statement `i *= 3 + 7` means `i = i * (3 + 7)`, not `i = i * 3 + 7`. For i = 5 the result is 50, not 22.

Example 6.1

In a shopping cart, the total starts at 0 TL. Two items, one of 45 TL and one of 30 TL, are added in turn, then a discount of 10 TL is applied, and finally the amount is divided into 2 installments. Which statements are written, and what is `total` after each step?

Statement	Computation	total
`int total = 0;`	initial value	0
`total += 45;`	0 + 45	45
`total += 30;`	45 + 30	75
`total -= 10;`	75 − 10	65
`total /= 2;`	65 / 2 (integer division)	32

The result in the last row is 32, not 32.5. Because `total` is an int, the division is integer division and the fraction is discarded. If the installment amount must be computed down to the hundredth of a lira, the variable must be chosen as `double` from the start. The quotient of two integers in Java is always an integer.

6.1.5 Identifier rules and naming

The names given to variables, methods, and classes are all identifiers. Java limits how an identifier may be written with a few strict rules. A name that violates these rules causes a compile error.

Rule	Valid	Invalid
Starts with a letter, _ or $	`student1`	`1student`
Contains no spaces	`totalScore`	`total score`
Is not a keyword	`classNo`	`class`
A single underscore cannot be a name	`_value`	`_`
Case-sensitive	`score` and `Score`	Assuming they are the same variable

Table 6.2: Identifier rules (the table in the slides).

An identifier starts with a letter, an underscore (_), or a dollar sign ($); the following characters can also include digits. A name that starts with a digit confuses the compiler. In the line `int 1student = 5;` the compiler expects a name after `int` but meets the digit 1 and cannot make sense of the line:

```
error: not a statement
int 1student = 5;
^
error: ';' expected
int 1student = 5;
   ^
2 errors
```

The messages do not state the problem directly; they say “this is not a statement” and “a semicolon was expected”. The compiler reports the error where it finds it, in the way it understands it. That is why, when reading a compiler message, you should look at the whole line the message points to.

Keywords are words that Java reserves for its own purposes: `int`, `double`, `class`, `if`, `final`, `new`, `return` are a few of them. These words cannot be used as names; the line `int class = 5;` produces three separate error messages. A lone underscore is not a name in JDK 21 either. The compiler rejects the line `int _ = 3;` with a message saying that unnamed variables are a preview feature. In newer Java versions `_` means “unnamed variable”; in that case too, the variable cannot be reached by its name.

The last rule is the most insidious. Java treats uppercase and lowercase letters as different characters. `score` and `Score` are two separate variables and can live side by side in the same program:

```
int score = 70;
int Score = 90;
System.out.println(score + " " + Score);
```

```
70 90
```

The program compiles without errors and prints two different values. If the programmer thinks they are using the same variable, the error arises silently. For this reason the use of upper and lower case is not left to chance; naming conventions provide this consistency.

Besides the rules the compiler checks, there are also unwritten **naming conventions** that Java programmers follow. Code that ignores them compiles but tires everyone who reads it.

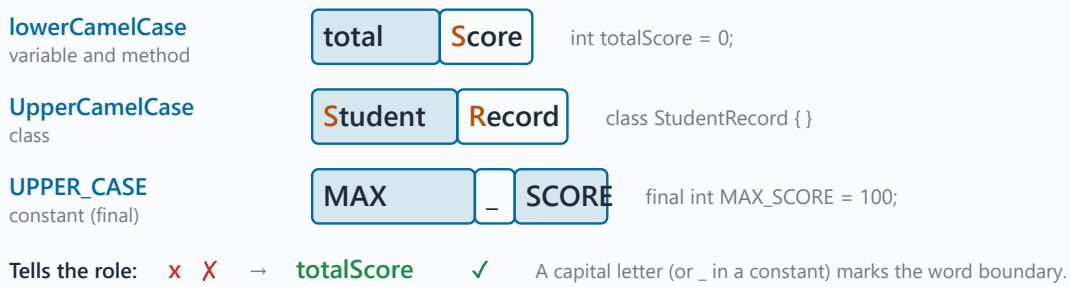


Figure 6.5: Three naming styles. Variable and method names start with a lowercase letter and every new word starts with a capital (`totalScore`). Class names start with a capital letter (`StudentRecord`). Constants are written in capital letters with words separated by underscores (`MAX_SCORE`).

- **lowerCamelCase**: variable and method names. The first word starts with a lowercase letter and the first letter of each following word is a capital: `totalScore` , `studentCount` , `computeAverage` . The capital letters between words show the word boundary, like the humps of a camel.
- **UpperCamelCase**: class names. Every word starts with a capital letter: `StudentRecord` , `Scanner` , `String` .
- **UPPER_CASE_WITH_UNDERSCORES**: `final` constants. `MAX_SCORE` , `VAT_RATE` .

The most important quality of a name is that it describes the role of the value. Names such as `x` , `a1` , and `temp2` make sense to the person writing the code at that moment; a week later they tell no one anything. The name `totalScore` , on the other hand, hints at both its type and its job. Non-ASCII letters, such as accented ones, are valid in Java identifiers, but in class and in most projects writing with ASCII letters is preferred: `student` , `grade` , `age` . This preference avoids problems that can arise when files move between computers with different character encodings.

Common mistake

Writing `totalScore` in one place and `totalscore` in another. If the second name has not been declared, the compiler gives a “cannot find symbol” error. Worse, if both names have been declared, the program compiles and the values go into different boxes. Following the naming conventions in the same way everywhere prevents most errors of this kind from the start.

6.2 Data types

At the lowest level, memory consists only of bits, that is, 0s and 1s. The bit pattern 01000001 can mean the number 65 in one place and the letter 'A' in another. The type determines how the bits are interpreted. Java requires the type of every variable to be known at compile time. That is why Java is a **statically typed** language: the compiler checks before the program runs whether every operation fits the types, and it catches many errors already at the first compilation.

6.2.1 The basic groups of types

Java types fall into two large families. **Primitive types** are eight basic types built into the language. In the box of a primitive variable sits the value itself. **Reference types** are strings (String), arrays, and classes. In the box of a variable of a reference type there is not the object itself but a **reference** to an object that sits somewhere else in memory.

Group	Example	Use
Integer	`byte` , `short` , `int` , `long`	Counting and discrete values
Floating-point	`float` , `double`	Measurement and approximate computation
Character	`char`	A single UTF-16 code unit
Logical	`boolean`	`true` / `false`
Reference	String, array, class	Reference to an object

Table 6.3: The basic groups of types (the table in the slides).

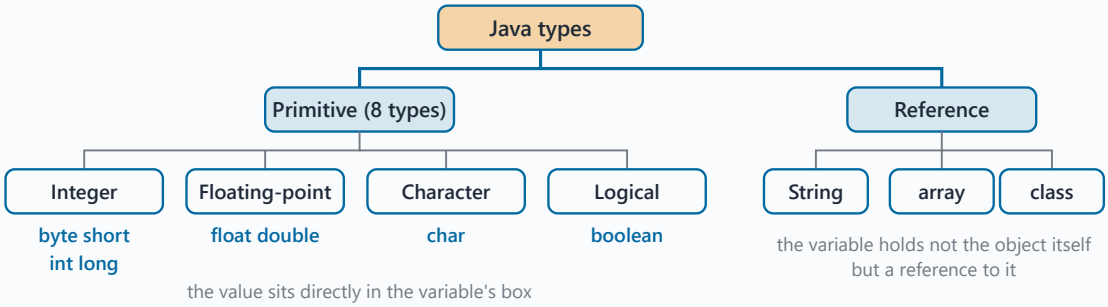


Figure 6.6: The two families of Java types. The eight primitive types are divided into four groups; in these, the value sits directly in the variable's box. String, arrays, and classes are reference types; the variable holds not the object itself but a reference to it.

The groups are chosen according to the nature of the data. **Counted** values, such as the number of students in a class, a position in an array, or a counter, are integers. **Measured** values, such as height, temperature, or price, are held as floating-point numbers. A single letter uses char, and a yes/no answer uses boolean. Texts such as names, addresses, and sentences are String objects. This chapter deals mainly with primitive types; String appears only in the context of concatenation and reading input.

6.2.2 Primitive types and their sizes

Each of the eight primitive types occupies a fixed number of bits in memory. This number is the same in Java on every computer and every operating system. In some languages, such as C, the size of int can vary from machine to machine; in Java, int is 32 bits everywhere. This consistency makes the same program give the same result everywhere.

Types	Bits	Smallest	Largest	Literal example
`byte`	8	−128	127	`byte level = 7;`
`short`	16	−32.768	32.767	`short year = 2024;`
`int`	32	−2.147.483.648	2.147.483.647	`int count = 42;`
`long`	64	−9.223.372.036.854.775.808	9.223.372.036.854.775.807	`long population = 8_000_000L;`
`float`	32	approximately $\pm 3.4 \times 10^{38}$; ≈ 7 significant digits		`float ratio = 1.5F;`
`double`	64	approximately $\pm 1.8 \times 10^{308}$; ≈ 15–16 significant digits		`double price = 19.90;`
`char`	16	0	65.535	`char grade = 'A';`
`boolean`	JVM-dependent	`true` or `false`		`boolean active = true;`

Table 6.4: The primitive types, their sizes, and their limits. The limit values were obtained by printing constants such as `Byte.MAX_VALUE` in JDK 21.

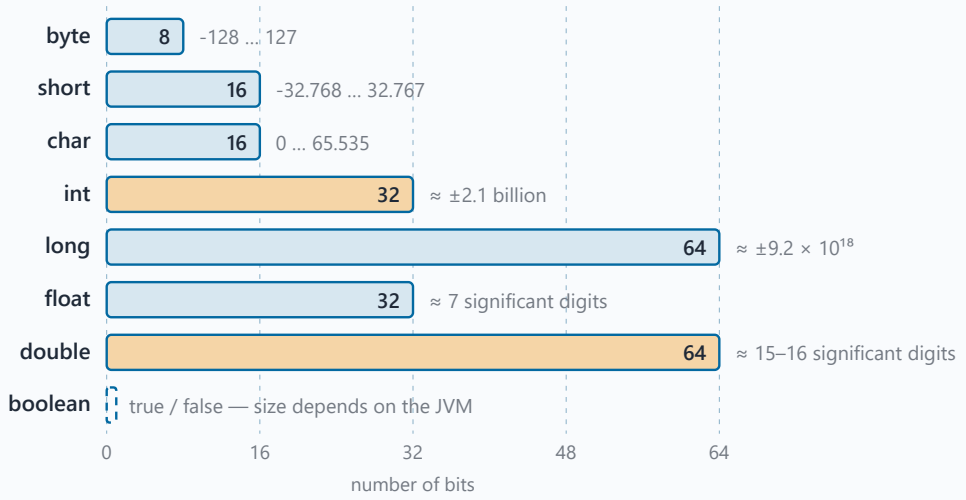


Figure 6.7: The bit counts of the primitive types. The highlighted bars are the default choices: int for integers and double for floating-point numbers. 1 byte = 8 bits.

The range of a type is computed from its number of bits. With n bits, 2^n different bit patterns can be written. For byte there are $2^8 = 256$ different values. Java's integer types are signed: half of these values go to negative numbers, and half go to zero and positive numbers. Because zero takes up a place on the positive side, the range extends from −128 to 127; the number 128 does not fit in a byte. The same computation gives the range $-2^{31} \ldots 2^{31} - 1$ for int:

> 2^{31} = 2,147,483,648, so the int range is −2,147,483,648 … 2,147,483,647

To keep it roughly in mind, the limit of int can be remembered as “about ±2.1 billion” and that of long as “about ±9.2 quintillion” (9.2×10^{18}). Although the char type is 16 bits, it is unsigned; it has no negative values and holds the codes from 0 to 65,535.

Values written directly in the program are called **literals**: `42`, `19.90`, `'A'`, `true`. Literals also have types, and the type can be told from the way they are written:

- An integer literal without a decimal point (`42`) is an int. If `L` is added at the end it becomes a long (`42L`). A lowercase `l` is also valid but is not used because it looks too much like the digit 1.
- A literal with a decimal point (`19.90`) or one written in scientific notation (`3.9e10`, that is, 3.9×10^{10}) is a double. If `F` is added at the end it becomes a float (`1.5F`).
- In long numbers, underscores can be used to group digits: `8_000_000_000L`. Underscores only make reading easier; they do not change the value.

In Java code the decimal separator is always a period. `19.90` is nineteen point ninety in Java; `19,90` is a syntax error. Code fragments and program outputs in this chapter keep exactly the form that Java writes.

Example 6.2

The world population (about 8 billion) is to be held in a variable. Why does the line `int population = 8_000_000_000;` not compile? Why does the line `long population = 8_000_000_000;` still not compile?

8 billion is larger than the largest int value, 2,147,483,647; the value does not fit in an int box. But the real problem starts earlier: the literal `8_000_000_000`, written without a decimal point and without an L, is itself of type int. While trying to read this literal as an int, the compiler exceeds the limit and reports an error without looking at the type on the left:

```
error: integer number too large
long population = 8_000_000_000;
                  ^
1 error
```

Even if the target is long, the message is the same. The correct way is to make the literal itself a long with L: `long population = 8_000_000_000L;`. This line compiles and `System.out.println(population)` prints 8000000000.

A similar rule exists for floating-point literals. `1.5` is a double literal; the line `float ratio = 1.5;` does not compile, with the message “possible lossy conversion from double to float”. When a float is wanted, the literal is written as `1.5F`. In most programs this question never arises, because the default choice is already double.

6.2.3 How are integers stored in memory?

When choosing an integer type, int is enough in most cases and is the default choice. long is used for large values that int cannot hold: the world population, time in milliseconds, a balance in hundredths of a lira in a bank account. byte and short are usually chosen to save memory in large data arrays. The largest value expected in a computation must be estimated beforehand; a value that may approach the limit of int is made long from the start. The importance of this decision lies in what happens when the limit is exceeded. To understand it, we need a brief look at how integers are represented with bits.

Java stores integers in **two's complement** representation. In this representation the leftmost bit is the **sign bit**: if it is 0 the number is zero or positive, and if it is 1 the number is negative. A few examples with byte:

Bits	Unsigned reading	byte value
`0000 0000`	0	0
`0000 0001`	1	1
`0111 1111`	127	127 (largest)
`1000 0000`	128	128 − 256 = −128 (smallest)
`1000 0010`	130	130 − 256 = −126
`1100 1000`	200	200 − 256 = −56
`1111 1111`	255	255 − 256 = −1

Table 6.5: Bit representation of some byte values. The value of a bit pattern whose sign bit is 1 is found by subtracting 256 from the unsigned value.

The last column of the table rests on a simple rule: an 8-bit pattern whose sign bit is 1 is read by subtracting 256 from its unsigned value. The same rule holds for int with 2^{32} and for long with 2^{64}. This rule is enough to compute overflow and narrowing conversions.

6.2.4 Overflow

What happens when the result of an integer operation does not fit in the range of the type? A car's odometer rolls over to 000,000 after 999,999. Java integers behave similarly: the result overflows at one end of the range and re-enters from the other end.

Definition 6.4

Overflow is when the result of an integer operation exceeds the range of the type. In Java, overflow does not cause an error. Only the low bits of the result that fit in the type are stored; the value silently wraps around to the other end of the range.

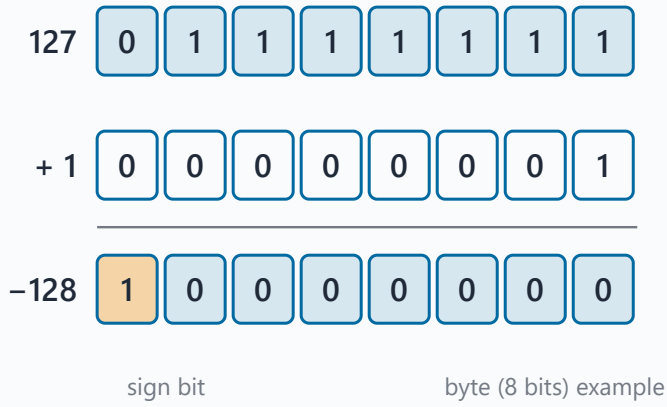


Figure 6.8: Overflow for byte. The bits of 127 are 0111 1111; adding 1 gives 1000 0000. Because the sign bit is 1, this pattern is read as −128.

The example program from the slides shows overflow in both types:

```
int x = Integer.MAX_VALUE;
x++;
byte b = 127;
b++;
System.out.println(x + " " + b);
```

```
-2147483648 -128
```

The program runs line by line as follows:

Line	Statement	x	b	Explanation
1	`int x = Integer.MAX_VALUE;`	2147483647	–	bits 0111 1111 … 1111
2	`x++;`	−2147483648	–	bits 1000 0000 … 0000
3	`byte b = 127;`	−2147483648	127	bits 0111 1111
4	`b++;`	−2147483648	−128	bits 1000 0000
5	`System.out.println(x + " " + b);`	−2147483648	−128	output: -2147483648 -128

Table 6.6: Trace table of the overflow example.



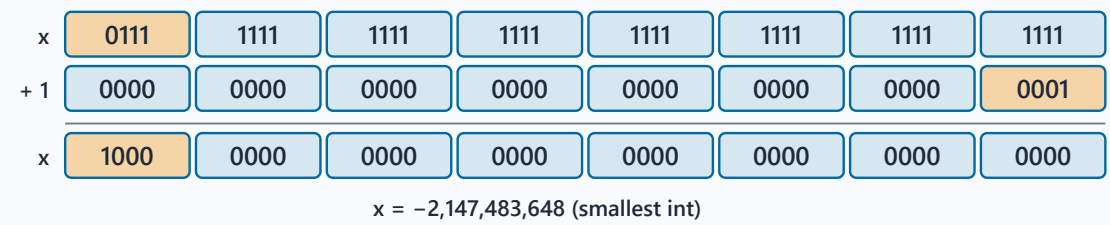


Figure 6.9: Overflow for int. In Integer.MAX_VALUE the sign bit is 0 and the other 31 bits are 1. When 1 is added, the carry travels through all the bits, the lower 31 bits become 0, and the sign bit turns to 1. The result is the smallest int: −2,147,483,648.

Binary addition works by the same carry logic as in the decimal system. When 1 is added to the pattern 0111…1, the sum of the rightmost 1 and 1 is 0 and a carry of 1 remains. The carry moves leftward, zeroing all the 1s, and finally reaches the sign bit. The result is 1000…0. Mathematically the correct answer is 2,147,483,648, but this value is outside the int range. The value of a pattern with sign bit 1 comes out as $2{,}147{,}483{,}648 - 2^{32} = -2{,}147{,}483{,}648$. In terms of the odometer analogy, the number overflows at the top of the range and re-enters at the bottom.

There is a subtlety with byte: the result of `byte + byte` is an int (6.3.5), so `b + 1` gives 128 and does not overflow. 8-bit overflow is seen only when the result is brought down to byte with a `(byte)` cast, or in compound operations such as `+=` and `++`; these operations perform the narrowing automatically.

The danger of overflow is that it is silent. The program does not stop and gives no warning; it carries on computing with the wrong number. Overflow does not occur only by adding one to the limit. Even the product of two reasonable-looking numbers can exceed int:

```
int a = 50000;
int b = 50000;
System.out.println(a * b);
System.out.println((long) a * b);
System.out.println(a * b / 1000);
long wrong = a * b;
System.out.println(wrong);
```

```
-1794967296
2500000000
-1794967
-1794967296
```

50,000 × 50,000 = 2,500,000,000 and exceeds the int limit. The first line therefore prints a negative number: $2{,}500{,}000{,}000 - 2^{32} = -1{,}794{,}967{,}296$. In the second line the conversion `(long) a` is made before the multiplication; the operation runs in long and the correct result comes out. The third line shows that overflow also contaminates later operations: the wrong product is divided by 1000 and a

wrong quotient is obtained. The fourth line is the most misleading. The result is assigned to a long variable, but the multiplication is still done in int, because the two **operands**, that is, the two values the operation acts on, are ints. The overflowed result is copied into the long. That the target is a long does not change the type of the operation.

> **Common mistake**
>
> Thinking that writing `long result = a * b;` makes the multiplication run in long. Java determines the type of an expression only by looking at its operands; it does not look at where the result will be assigned. The product of two ints is an int and overflows if necessary. The correct way is to make one of the operands a long before the operation: `long result = (long) a * b;` or `a * 1L * b`.

6.2.5 Limit constants and Math.addExact

Writing the limit values of types by hand is both hard and dangerous; a single digit in the number 2147483647 can be mistyped. Java offers these values as ready-made constants:

Constant	Value
`Integer.MIN_VALUE`	-2147483648
`Integer.MAX_VALUE`	2147483647
`Long.MIN_VALUE`	-9223372036854775808
`Long.MAX_VALUE`	9223372036854775807

Table 6.7: The limit constants of int and long (the table in the slides).

Similar constants exist for the other types too: `Byte.MAX_VALUE`, `Short.MIN_VALUE`, `Double.MAX_VALUE`. These names are both readable and impossible to mistype.

If you do not want overflow to pass silently, the "exact" methods of the `Math` class are used. `Math.addExact`, `Math.subtractExact`, and `Math.multiplyExact` perform the operation; if the result does not fit in the type, instead of silently wrapping around they throw an **exception**. An exception is an error notification that interrupts the normal flow of the program.

```
int x = Integer.MAX_VALUE;
System.out.println(x + 1);
// -2147483648 (silent overflow)
System.out.println(
    Math.addExact(x, 1));
// ArithmeticException:
// integer overflow
```

```
-2147483648
Exception in thread "main" java.lang.ArithmeticException: integer overflow
```

The first `println` shows the silent overflow and prints −2147483648. In the second line `Math.addExact(x, 1)` is called; because the result does not fit in an int, an `ArithmeticException` is thrown and the program stops with the message “integer overflow”. For long values the message is “long overflow”. The program stopping looks bad at first glance, but it is much better than continuing with a wrong result: the error is noticed immediately and its location is known.

A surprising example of overflow is the absolute value. `Math.abs(Integer.MIN_VALUE)` gives −2147483648 again, instead of the expected 2,147,483,648. The cause is that the range is not symmetric: 2^{31}, the absolute value of the smallest int, does not fit in an int. Taking its negative causes overflow and the number comes back to itself. The same happens with the expression `-Integer.MIN_VALUE`.

6.2.6 Floating-point numbers

Numbers with a fractional part are stored in Java as **floating-point numbers**. The name comes from the number being held in a form similar to scientific notation: a series of significant digits and an exponent that indicates where the point goes. In the notation $12{,}345.678 = 1.2345678 \times 10^4$, the point “floats” according to the exponent. This lets the same 64 bits hold both very small (10^{-300}) and very large (10^{300}) numbers.

double is the default type for most approximate computation; it has about 15–16 significant digits. float is half the size and holds about 7 significant digits. The difference is easy to see:

```
System.out.println(0.1 + 0.2);
System.out.println(1.0 - 0.9);
System.out.println(0.1 + 0.2 == 0.3);
System.out.println(0.1 + 0.2 - 0.3);
System.out.println(Math.abs(0.1 + 0.2 - 0.3) < 1e-9);
System.out.println(0.1f + 0.2f);
System.out.println(1.1f * 3);
System.out.println(1.1 * 3);
System.out.println(100.0 / 3);
System.out.println(100.0f / 3);
System.out.println(0.5 + 0.25 == 0.75);
System.out.println(new java.math.BigDecimal(0.1));
System.out.println(new java.math.BigDecimal(0.2));
System.out.println(new java.math.BigDecimal(0.1 + 0.2));
System.out.println(new java.math.BigDecimal(0.3));
```

```
0.30000000000000004
0.09999999999999998
false
5.551115123125783E-17
true
0.3
3.3000002
3.3000000000000003
33.333333333333336
33.333332
true
0.1000000000000000055511151231257827021181583404541015625
0.200000000000000011102230246251565404236316680908203125
0.300000000000000044408920985006261616945266723632812 5
0.29999999999999998889776975374843459576368331909179 6875
```

When the lines of this output are read one by one, the basic problem of floating-point numbers appears. The problem is that the binary base cannot represent some decimal fractions exactly. In the decimal system 1/3 is an infinite fraction: 0.3333... No finite number of digits can write it exactly. In the binary system the same problem occurs for 1/10. The binary expansion of 0.1 goes on forever as 0.0001100110011... The 52-bit fraction field in memory cuts this expansion off at some point, and the number is stored as a value very close to 0.1 but slightly different from it.

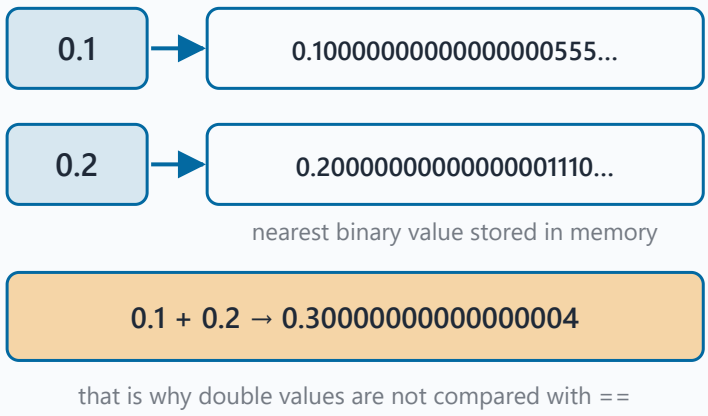


Figure 6.10: 0.1 and 0.2 cannot be stored exactly in memory; the nearest binary values are stored. When these small errors are added, the result prints as 0.30000000000000004.

The last four lines of the output show, with the `BigDecimal` class, the exact decimal expansion of the values in memory. The stored "0.1" is in reality 0.1000000000000000055511.... The stored "0.2" is also slightly larger than 0.2. When the two small excesses are added, the result falls not on the nearest double to 0.3 but on the next one. `println` writes this value as 0.30000000000000004. The value stored for 0.3 is 0.29999999999999988889.... Because the two numbers differ, the expression `0.1 + 0.2 == 0.3` gives false.

The same lines carry two more lessons. `1.1f * 3` gives 3.3000002 in float and 3.3000000000000003 in double. The error in float shows up in the seventh digit, and in double in the sixteenth. On the other hand, `0.5 + 0.25 == 0.75` is true. Because 0.5 and 0.25 can be written with powers of two (1/2 and 1/4), they are stored exactly. The source of the error is not floating-point numbers themselves but the fact that decimal fractions expand infinitely in binary.

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

```
0.30000000000000004
false
true
```

This short program from the slides shows the right way to compare. Two floating-point numbers are not compared with `==`. Instead, the absolute value of their difference is compared with a small **tolerance** (epsilon): `Math.abs(a - 0.3) < 1e-9`. The difference here is about 5.55×10^{-17} and is far smaller than 10^{-9}; the comparison gives true. The size of the tolerance is chosen according to the problem. For measurements at the millimeter level, 10^{-6} is enough; for very small physical quantities a smaller tolerance is needed.

Small errors accumulate in repeated operations. A loop that adds 0.1 ten times finds 0.9999999999999999, not 1.0. This accumulation is unacceptable in computations such as money, where every cent must be exact. In such computations, either the amounts are held as integers (long) in hundredths of a lira, or the `BigDecimal` class is used. BigDecimal stores the number with its decimal digits:

```
BigDecimal a = new BigDecimal("0.1");
BigDecimal b = new BigDecimal("0.2");
System.out.println(a.add(b));
System.out.println(a.add(b).compareTo(new BigDecimal("0.3")) == 0);
BigDecimal price = new BigDecimal("19.90");
BigDecimal qty = new BigDecimal("3");
System.out.println(price.multiply(qty));
```

```
0.3
true
59.70
```

A BigDecimal object is created from a string (`"0.1"`); if `new BigDecimal(0.1)` is written, the inexact double value is copied as it is. Addition is done with `add` instead of `+`, and multiplication with the `multiply` method. The cost of three items of 19.90 TL comes out as exactly 59.70.

> **Common mistake**
>
> Comparing two double values with `==` and being surprised: the condition `if (total == 1.0)` is never satisfied if total was built up by accumulating 0.1s. Floating-point numbers are always compared with a tolerance. Also, money computations are not done with double; integer hundredths or BigDecimal are used.

6.2.7 char and boolean

The char type holds a single character and its literal is written with single quotes: `'A'`, `'7'`, `'ç'`. A computer cannot store letters directly; each character corresponds to a number, and that number sits in the box. This mapping is the job of the Unicode standard. The code of the letter 'A' is 65, that of 'a' is 97, that of the digit '0' is 48, and that of the space character is 32. A Java char is a **code unit** of the UTF-16 encoding, which stores Unicode text in 16-bit units. All the Turkish letters fit in a single char: 'ç' is stored with code 231, 'ğ' 287, 'ı' 305, 'İ' 304, and 'ş' 351. (Very rarely used characters, such as some emoji, need two chars; this detail does not come up in beginner programs.)

Single quotes and double quotes produce different types. `'A'` is a char; the number 65 sits in the box. `"A"` is a one-letter **string**, that is, a String object; the variable holds a reference to this object. The two cannot be used in place of each other. The line `char c = "A";` does not compile, with the message “String cannot be converted to char”.

The boolean type takes only two values: `true` and `false`. The results of comparisons and conditions are booleans. In C, 0 counts as false and every nonzero number counts as true; Java has no such mapping. The line `boolean active = 1;` gives an “int cannot be converted to boolean” error. This strictness catches a common mistake in conditions at compile time.

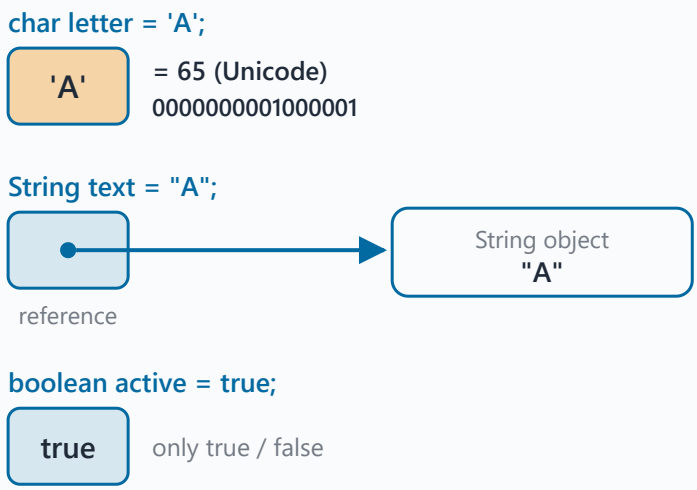


Figure 6.11: Three different boxes. The char box holds 65, the code of the letter 'A' (16 bits). The String variable holds a reference to an object in memory. The boolean box takes only true or false.

6.2.8 char arithmetic

Because a char box holds a number, arithmetic can be done with char values. The example from the slides shows this relationship step by step:

```
char c = 'A';
c++;
int k = c + 1;
char d = (char)(c + 1);
System.out.println(c + " " + k + " " + d);
```

```
B 67 C
```

Line	Statement	c (char)	k (int)	d (char)	Explanation
1	`char c = 'A';`	'A' (65)	–	–	the box holds 65
2	`c++;`	'B' (66)	–	–	the code increases by 1; the type stays char
3	`int k = c + 1;`	'B' (66)	67	–	`c + 1` is an int: 66 + 1 = 67
4	`char d = (char)(c + 1);`	'B' (66)	67	'C' (67)	67 is converted back to char
5	`System.out.println(…);`	'B'	67	'C'	output: B 67 C

Table 6.8: Trace table of the char arithmetic example. The numeric code in the box is given in parentheses.

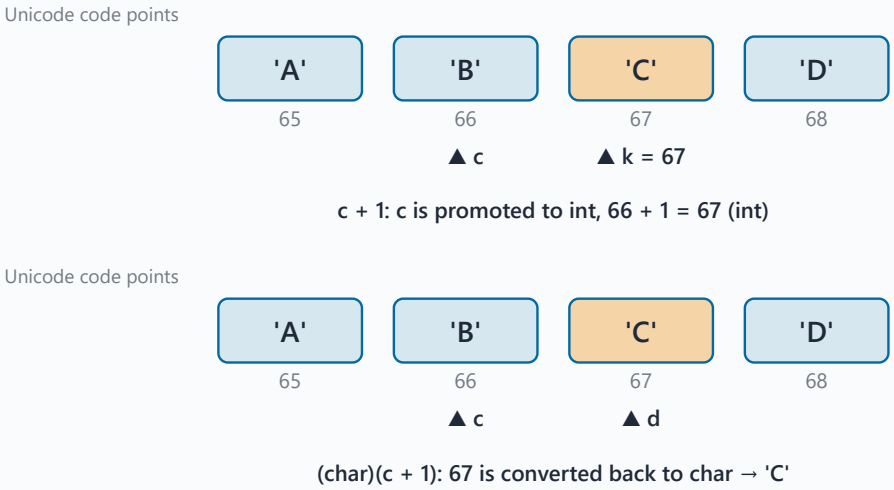


Figure 6.12: c, k, and d on the line of Unicode code points. In the expression `c + 1`, c is promoted to int and the result is the number 67. The conversion `(char)(c + 1)` interprets the same 67 as a letter again: 'C'.

The crucial point of the example is the third line. In the expression `c + 1`, a char and an int are added. In arithmetic operations Java first converts the char to int; the result is an int. That is why `k` holds the number 67, not the letter 'C'. To store the result as a letter, an explicit `(char)` cast is needed. If the cast is not written, the line does not compile:

```
char c = 'A';
char d = c + 1;
```

```
error: incompatible types: possible lossy conversion from int to char
char d = c + 1;
          ^
1 error
```

The `c++` on the second line, however, compiles without problems; the `++` operator preserves the variable's own type. On the last line, printing is done according to type: the char value appears as a letter (B, C), and the int value appears as a number (67).

char arithmetic has practical uses:

```
System.out.println((int) 'A' + " " + (int) 'a' + " " + (int) '0' + " " + (int) ' ');
System.out.println('a' - 'A');
System.out.println((char) ('a' - 32));
System.out.println('7' - '0');
System.out.println('A' + 'B');
System.out.println("" + 'A' + 'B');
System.out.println((int) 'ç' + " " + (int) 'ğ' + " " + (int) 'İ' + " " + (int) 'ı' + " " +
(int) 'ş');
char c = 'A';
c += 2;
System.out.println(c);
char h = 66;
System.out.println(h);
```

```
65 97 48 32
32
A
7
131
AB
231 287 304 305 351
C
B
```

The difference between lowercase and uppercase letters is constant in the English alphabet: *'a' − 'A' = 97 − 65 = 32*. When 32 is subtracted from a lowercase letter and the result is converted to char, the uppercase letter is obtained. When '0' is subtracted from a digit character, the numeric value of the digit is found: *'7' − '0' = 55 − 48 = 7*. This method is used to process, one by one, the digits of a number that has been read as text. This shortcut does not work for Turkish letters: the difference between 'ı' (305) and 'I' (73) is not 32. For case conversion in Turkish text, Java's ready-made locale-aware methods are preferred.

The fifth and sixth lines of the output show a trap. In the expression `'A' + 'B'`, two chars are added; both are converted to int and the result is the number *65 + 66 = 131*, not "AB". To write letters side by side, the operation has to start with a string: `"" + 'A' + 'B'` gives "AB". The rule is simple: if one of the operands of the `+` operation is a string, the operation is concatenation, not addition. The last two lines show how char keeps its type. `c += 2` advances 'A' to 'C'; compound assignment converts the result to char automatically. The line `char h = 66;` also compiles, because the literal 66 is within the char range and the compiler finds this conversion safe; h holds the letter 'B'.

6.3 Type conversion

Different types often meet in a program. A number of students (int) is multiplied by an average (double), a number (int) is added to the code of a letter (char), a double result is to be written into an int box. At every meeting, Java has to convert a value from one type to another. This operation is called **type conversion**. Java performs some conversions on its own, while others require the programmer to ask for them explicitly. The criterion for the distinction is a single one: is there a risk of losing information in the conversion?

6.3.1 Assignment compatibility: widening and narrowing

Definition 6.5

A **widening conversion** converts a value to a type with a wider range (for example from int to double). Java performs widening automatically, that is, **implicitly**. A **narrowing conversion** converts to a type with a narrower range (for example from double to int). Because narrowing can lose information, it requires an **explicit conversion** (cast): the target type is written in parentheses in front of the value, `(int) 9.8`.

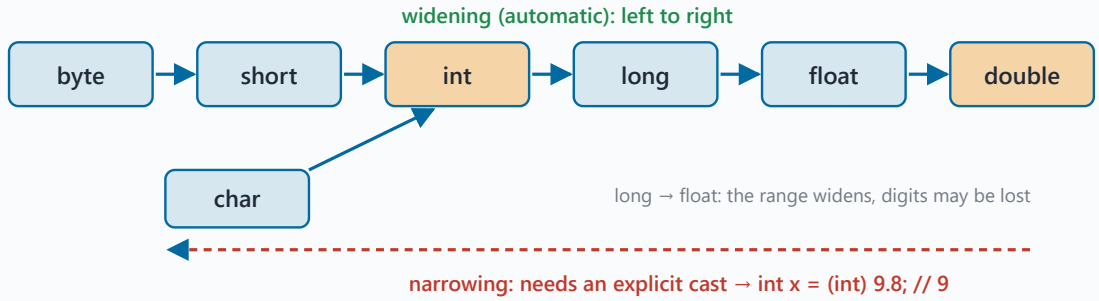


Figure 6.13: The widening chain: byte → short → int → long → float → double. char also widens to int. Conversions from left to right are automatic; the reverse direction is narrowing and requires an explicit cast.

Widening is safe, because every value of the narrow type also exists in the wide type. The number 7 can be represented in both int and double; when converted to double it becomes 7.0 and nothing is lost. All of the following assignments compile:

```
int count = 7;
double wide = count;
long asLong = count;
float f = 10;
double d = 'A';
long l = 5;
System.out.println(wide + " " + asLong + " " + f + " " + d + " " + l);
long large = 123456789123L;
float ff = large;
System.out.println(ff);
int i2 = 123456789;
float f2 = i2;
System.out.println(f2 + " " + (int) f2);
```

```
7.0 7 10.0 65.0 5
1.2345679E11
1.2345679E8 123456792
```

The values in the first line show the expected conversions: int 7 becomes 7.0 in double and 7 in long; 10 becomes 10.0 in float; 'A' becomes 65.0 in double. At one link of the chain, however, there is a subtle loss. The step from long to float and from int to float counts as “widening” because the range widens. But float holds only about 7 significant digits. When the number 123,456,789 is converted to float and back, 123,456,792 comes out; the last digits have been rounded. Java gives no warning for this conversion. When working with large integers, choosing double instead of float largely removes the problem.

In narrowing, however, loss of information is the normal case. The double 9.8 does not fit in an int box; the fractional part has to be dropped. Java leaves this decision to the programmer and refuses the assignment unless an explicit cast is written:

```
double ratio = 9.8;
int narrow = ratio;
```

```
error: incompatible types: possible lossy conversion from double to int
int narrow = ratio;
             ^
1 error
```

The phrase “possible lossy conversion” in the message means that the conversion might lose information. The compiler does not know whether a loss will actually happen; it reports only that it could. When `int narrow = (int) ratio;` is written, the programmer says explicitly that the loss is

accepted, and the line compiles. The same rule applies to literals: the line `int x = 5L;` gives "possible lossy conversion from long to int" even though the number 5 fits easily in an int, because the type of the literal is long.

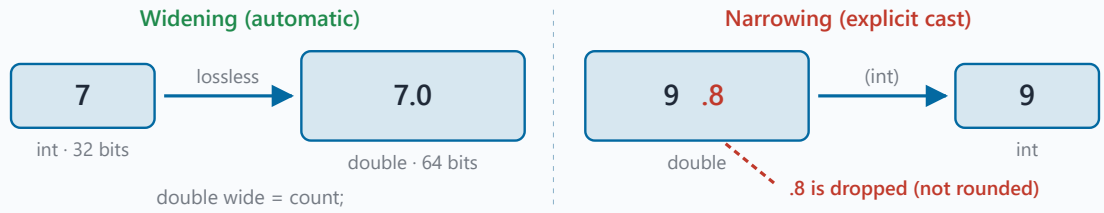


Figure 6.14: Widening and narrowing. int 7 passes into the double box losslessly as 7.0. double 9.8 is converted to 9 with `(int)`: .8 is dropped, and no rounding is done.

A conversion does not change the source variable. The expression `(int) ratio` reads the 9.8 in the `ratio` box and produces the value 9; `ratio` is still 9.8. The conversion affects only the result it produces. Thinking that a conversion permanently changes a variable's type is a common misconception; in Java the type of a variable is determined at the moment it is declared and never changes afterwards.

6.3.2 Type conversion step by step

The example from the slides shows widening, narrowing, and the two forms of division in a single program:

```
int count = 7;
double wide = count;       // 7.0
double ratio = 9.8;
int narrow = (int) ratio;     // 9
int quot = 5 / 2;          // 2
double exact = 5.0 / 2;    // 2.5
```

Line	Statement	New box	Value	What happened?
1	`int count = 7;`	count (int)	7	32-bit int box
2	`double wide = count;`	wide (double)	7.0	widening: automatic, lossless
3	`double ratio = 9.8;`	ratio (double)	9.8	64-bit double box
4	`int narrow = (int) ratio;`	narrow (int)	9	narrowing: the fraction was dropped; ratio is still 9.8
5	`int quot = 5 / 2;`	quot (int)	2	int / int: integer division
6	`double exact = 5.0 / 2;`	exact (double)	2.5	double / int: 2 widens to 2.0

Table 6.9: Trace table of the type conversion example. Each row shows only the newly allocated box; the earlier ones do not change.

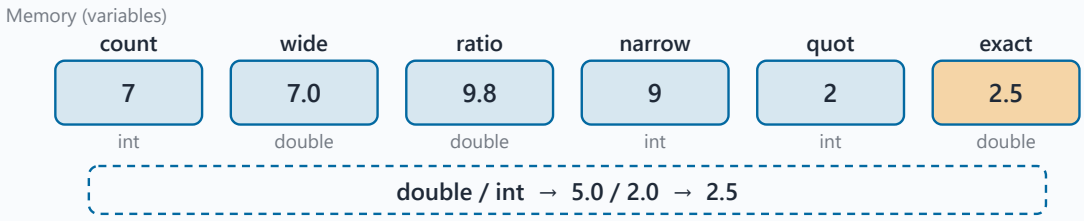


Figure 6.15: Memory at the end of the program. Five of the six boxes were filled on earlier lines; on the last line *5.0 / 2.0 → 2.5* is computed and written into the `exact` box.

The conversion on the fourth line is **truncation**, not rounding: 9.8 is converted not to the nearest integer, 10, but to 9. Whatever the decimal part, it is dropped; 9.99 also becomes 9. The fifth and sixth lines preview the topic of the next section: the same numbers 5 and 2 give the result 2 or 2.5 depending on how they are written.

6.3.3 Overflow and truncation in conversions

An explicit conversion tells the compiler, "I accept the loss." How much will be lost depends on the kind of conversion. The table in the slides gives the commonly encountered cases together. The values were verified by running them on JDK 21.

Expression	Result	Why
`(byte) 200`	-56	only the low 8 bits remain: 200 − 256
`(int) 3000000000L`	-1294967296	long → int: the upper bits are discarded
`(int) 3.9`	3	the decimal part is cut off, not rounded
`(int) -2.7`	-2	truncated toward zero
`(int) 3.9e10`	2147483647	double → int: clamps at the int limit
`Math.round(3.9)`	4	nearest integer (returns a long)
`Math.round(-2.7)`	-3	nearest integer
`Math.round(3.9e10)`	39000000000	fits because it is a long

Table 6.10: Results of narrowing conversions and of Math.round (the table in the slides).

The rows of the table show three different behaviors.

Integer to a narrower integer. The conversion keeps only the low bits of the source number that fit in the target type and discards the upper bits. The binary representation of 200 is 1100 1000. These eight bits are copied into the byte as they are. Because the sign bit is 1, the value is read as *200 − 256 = −56*. For 3,000,000,000 the computation is done at int size: *3,000,000,000 − 2^{32} = 3,000,000,000 − 4,294,967,296 = −1,294,967,296*. This is the same as the wrap-around rule in overflow. A few more examples: `(byte) 130` gives −126, `(byte) 256` gives 0, `(byte) -129` gives 127, and `(short) 40000` gives −25536.

Floating-point to integer. The decimal part is dropped and the number is truncated **toward zero**: positive numbers go down and negative numbers go up. That is why 3.9 → 3 and −2.7 → −2. It may be surprising that −2.7 goes to −2 and not to −3; −2 is greater than −3 and closer to zero. If the number exceeds the range of the target type, the bits are not cut off; the result is clamped at the end of the range. 3.9×10^{10} does not fit in an int and 2147483647, which is Integer.MAX_VALUE, comes out. Likewise `(int) -3.9e10` gives the smallest int, and `(int) Double.NaN` gives 0.

Rounding. If the nearest integer is wanted instead of truncation, `Math.round` is used. 3.9 → 4 and −2.7 → −3. Math.round returns a long when given a double; that is why 3.9×10^{10} fits without problems. If the result is to be written into an int box, a second conversion is needed: `int ri = (int) Math.round(x);`. If the cast is not written, the line `int r = Math.round(2.7);` gives a "possible lossy conversion from long to int" error.

> **Common mistake**
>
> Thinking that the `(int)` cast rounds the number. `(int) 9.8` is 9, not 10; `(int) -2.7` is −2, not −3. For rounding, write `Math.round`, and for an int result, `(int) Math.round(x)`. A narrowing conversion does not cause an error; the data is silently lost.

6.3.4 Integer division: 5 / 2 versus 5.0 / 2

In Java the division operator `/` does two different jobs and decides which to do by looking at the types of the operands. If both operands are integers, **integer division** is performed: the result is the whole part of the quotient and the remainder is discarded. If at least one of the operands is floating-point, the other is promoted to floating-point as well and ordinary division is performed.

```
System.out.println(5 / 2);
System.out.println(5.0 / 2);
System.out.println(5 / 2.0);
System.out.println((double) 5 / 2);
System.out.println((double) (5 / 2));
System.out.println(5 % 2);
```

```
2
2.5
2.5
2.5
2.0
1
```

Expression	Operand types	Operation performed	Output
`5 / 2`	int, int	integer division	2

Expression	Operand types	Operation performed	Output
`5.0 / 2`	double, int	2 → 2.0; 5.0 / 2.0	2.5
`5 / 2.0`	int, double	5 → 5.0; 5.0 / 2.0	2.5
`(double) 5 / 2`	double, int	conversion first: 5.0; then 5.0 / 2.0	2.5
`(double) (5 / 2)`	int, int	first 5 / 2 = 2; then 2 → 2.0	2.0
`5 % 2`	int, int	remainder: 5 = 2 · 2 + 1	1

Table 6.11: Operand types and results of the six lines.

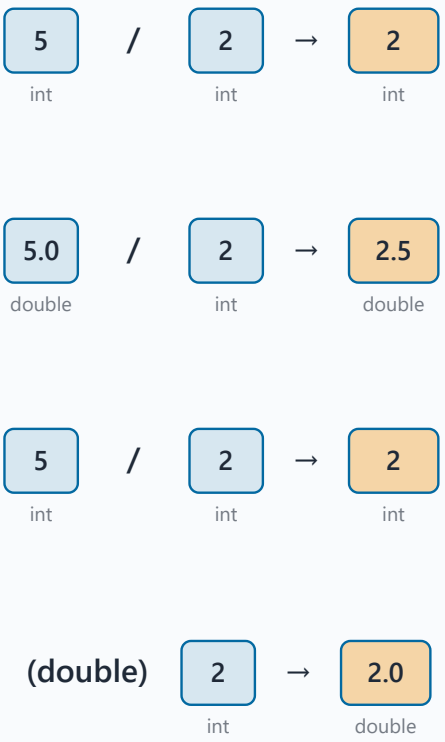


Figure 6.16: The operand types of the division expressions. Two ints: the result is 2 (int). One double: the int operand widens to double and the result is 2.5. In the expression `(double) (5 / 2)`, the inside of the parentheses is first evaluated as an int (2), and the conversion comes only afterwards (2.0).

The difference between the fourth and fifth lines rests on a single pair of parentheses, but the result changes. A type cast is a unary operator that comes before the division. In the expression `(double) 5 / 2`, 5 is first converted to double and then the division is done in floating-point. In the expression `(double) (5 / 2)`, the parentheses are evaluated first; the division is done between two ints and the 0.5 is lost. The later conversion turns 2 into 2.0 but does not bring back the lost fraction.

Example 6.3

The average of the scores 70, 85, and 90 is computed with four different expressions. Which gives the correct result?

```
int n1 = 70, n2 = 85, n3 = 90;
System.out.println((n1 + n2 + n3) / 3);
System.out.println((n1 + n2 + n3) / 3.0);
System.out.println((double) (n1 + n2 + n3) / 3);
System.out.printf("%.2f%n", (n1 + n2 + n3) / 3.0);
```

```
81
81.66666666666667
81.66666666666667
81.67
```

The sum is *70 + 85 + 90 = 245*. In the first line, because 245 and 3 are ints, integer division is performed: *245 / 3 = 81*, and the remainder 2 is discarded. In the second line 3.0 is a double; 245 is promoted to double as well and *245 / 3 = 81.666...* is found. In the third line the cast is made before the division and the result is again correct. The last line prints the result with two decimal places: 81.67. For a correct average, at least one operand of the division must be floating-point.

Integer division is not always an error; much of the time it is exactly the operation wanted. If 17 apples are put into packages of 5, *17 / 5 = 3* full packages result. In 3725 seconds there is *3725 / 3600 = 1* full hour. In such computations, integer division is used together with the `%` operator, which gives the remainder. The mistake is dividing two integers when a fractional result such as an average or a ratio is expected.

6.3.5 Type promotion

The rule seen in division holds for all binary arithmetic operators.

Definition 6.6

Type promotion is the automatic conversion of a narrower operand to the wider type before the operation in an expression of mixed types. The result is of the wider type: int + double → double, int * long → long. In addition, byte, short, and char operands are first promoted to int in every arithmetic operation.

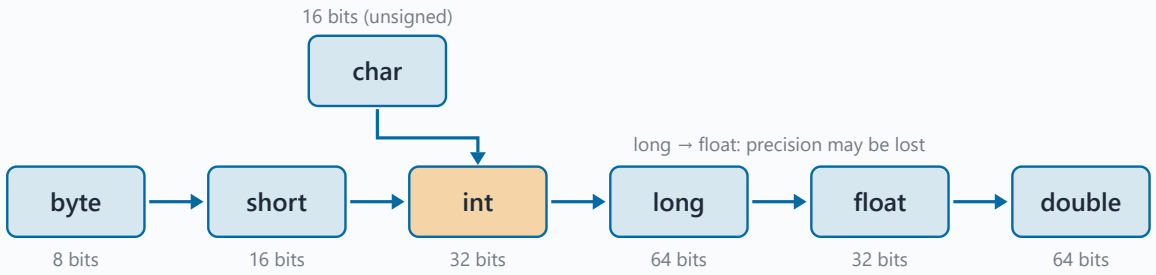


Figure 6.17: The type promotion order. Conversions in the direction of the arrows are automatic; in the opposite direction an explicit cast is needed. char is a 16-bit unsigned type and is promoted to int. Precision may be lost in the long → float step.

The second half of the rule is surprising: even the sum of two bytes is an int. Java does no arithmetic directly on these small types; processors work more efficiently with 32-bit operations, and the risk of overflow in the intermediate result is reduced. As a result, the following lines do not compile:

```
short a = 1, b = 2;
short c = a + b;
```

```
error: incompatible types: possible lossy conversion from int to short
short c = a + b;
          ^
1 error
```

`a + b` is an int, and narrowing from int to short requires an explicit cast: `short c = (short) (a + b);`. For the same reason `byte b = 10; b = b + 5;` does not compile either. This behavior also explains why the expression `c + 1` seen in the char arithmetic is an int.

If an expression has several operations, promotion is done separately for each operation. In the expression `1 / 2 * 10.0`, first `1 / 2` is computed between two ints and gives 0; only then, when 0 is multiplied by 10.0, is it promoted to double. The result is 0.0, not 5.0. The presence of a double operand somewhere in an expression does not make the whole expression floating-point; promotion happens only in the operation in which that operand takes part.

6.4 Operators

Definition 6.7

An **operator** is a symbol that acts on one or more values: `+`, `*`, `<`, `&&`. The values an operator acts on are called **operands**. A combination of operators, variables, and literals that produces a value is called an **expression**: `a + b * 2`.

Operators are named according to the number of operands they take. Operators with a single operand, such as `-x`, `!active`, and `i++`, are **unary**; those with two operands, such as `a + b`, are **binary** operators. Every expression has a value and a type. The value of the expression `7 / 2` is 3 and its type is int; the value of the expression `7 > 2` is true and its type is boolean.

6.4.1 Arithmetic operators and the remainder

Operator	Meaning	Example
`+ -`	Addition / subtraction	`a + b`
`* /`	Multiplication / division	`a * b`
`%`	Remainder	`num % 2`
`++ --`	Increment / decrement by one	`i++`
`+= -=`	Compound assignment	`total += value`

Table 6.12: Arithmetic operators (the table in the slides).

Addition, subtraction, and multiplication are as in mathematics; the only difference is that the result has to fit in the range of the type. The two faces of the division operator were seen in the section on integer division. The main topic of this section is the `%` operator.

Definition 6.8

The **remainder** operator `%` gives what is left over from integer division. For integers a and b, the equality $a = (a / b) \cdot b + (a \% b)$ always holds. In a division, the number being divided is called the **dividend** and the whole part of the result is called the **quotient**.

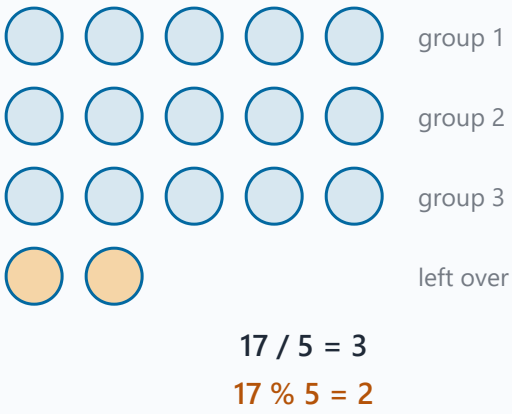


Figure 6.18: 17 objects grouped in fives. 3 full groups form (*17 / 5 = 3*) and 2 objects are left over (*17 % 5 = 2*).

Figure 6.18 shows the grouping of 17 objects in fives: `/` tells how many full groups form, and `%` tells how many objects are left over. The equality can be checked: *3 · 5 + 2 = 17*. The remainder operator is used much more often than one might think at first:

- **Even-odd test.** If `num % 2 == 0`, the number is even. In general, `a % b == 0` shows that a is divisible by b.
- **Separating digits.** `num % 10` gives the last digit; `num / 10` drops the last digit. For 1234, *1234 % 10 = 4* and *1234 / 10 = 123*.
- **Unit conversion.** Splitting seconds into minutes and seconds, or an amount in hundredths of a lira into lira and hundredths: for 1575 hundredths, *1575 / 100 = 15* lira and *1575 % 100 = 75* hundredths.
- **Cyclic counting.** Values that wrap around, such as the day of the week or a clock face: if today is Wednesday (2), then 10 days later is *(2 + 10) % 7 = 5*, that is, Saturday (taking Monday as 0).

For negative numbers the sign of the remainder follows a rule: in Java the remainder always takes the sign of the **dividend**. This rule follows from integer division truncating toward zero. The first four lines of the following program test negative operands, and the later lines test the other cases that came up in this section and in the section on division by zero:

```
System.out.println(7 / 2 + " " + 7 % 2);
System.out.println(-7 / 2 + " " + -7 % 2);
System.out.println(7 / -2 + " " + 7 % -2);
System.out.println(-7 / 3 + " " + -7 % 3 + " " + Math.floorMod(-7, 3) + " " +
Math.floorDiv(-7, 3));
System.out.println(17 / 5 + " " + 17 % 5);
System.out.println(5.5 % 2);
System.out.println(1 / 2 * 10 + " " + 10 * 1 / 2);
System.out.println(1.0 / 0 + " " + -1.0 / 0 + " " + 0.0 / 0);
double nan = 0.0 / 0;
System.out.println((nan == nan) + " " + (nan != nan) + " " + Double.isNaN(nan));
System.out.println(Math.sqrt(-1) + " " + (1.0 / 0 - 1.0 / 0) + " " + (1.0 / 0 + 5));
System.out.println(1 / 0.0 > 1e308);
```

```
3 1
-3 -1
-3 1
-2 -1 2 -3
3 2
1.5
0 5
Infinity -Infinity NaN
false true true
NaN NaN Infinity
true
```

The first four lines of the output show this rule. *−7 / 2 = −3* (−3.5 is truncated toward zero), and the remainder is *−7 − (−3 · 2) = −1*. *7 / −2 = −3* and *7 % −2 = 1*; because the dividend 7 is positive, the remainder is positive too. *−7 % 3 = −1*; the "remainder between 0 and 2" customary in mathematics is 2. If that value is needed, `Math.floorMod(-7, 3)` is used. That is why whether a number is odd is tested not with `num % 2 == 1` but with `num % 2 != 0`; for −7 the first gives a wrong result.

The fifth line is the 17-object example from the figure. The % operator also works with floating-point numbers: since *5.5 = 2 · 2 + 1.5*, `5.5 % 2` gives 1.5. The seventh line of the output shows the effect of evaluation order on integer division: `1 / 2 * 10` first computes *1 / 2 = 0* and gets 0; `10 * 1 / 2` first computes *10 · 1 = 10* and gets 5. In mathematics the two expressions are equal; in Java they are not, because of integer division.

6.4.2 Division by zero and special values

In mathematics, division by zero is undefined. Java treats this case differently for integers and for floating-point numbers.

Expression	Result	Explanation
`1 / 0`	ArithmeticException: / by zero	division by zero in integer division is a runtime error

Expression	Result	Explanation
`1.0 / 0`	Infinity	no error with decimals: infinity
`-1.0 / 0`	-Infinity	negative infinity
`0.0 / 0`	NaN	Not a Number: an undefined result
`NaN == NaN`	false	NaN is not even equal to itself
`NaN != NaN`	true	to test for NaN use `Double.isNaN(x)`
`-7 % 3`	-1	the sign of the remainder is the sign of the dividend
`5.5 % 2`	1.5	% also works with floating-point numbers

Table 6.13: Division by zero and special values (the table in the slides).

Division by zero with integers stops the program. Compilation succeeds, because the compiler often cannot know in advance that the divisor will be zero. The error appears while the program runs:

```
int a = 1;
int b = 0;
System.out.println("before");
System.out.println(a / b);
System.out.println("after");
```

```
before
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

“before” is printed, the JVM throws an `ArithmeticException` at the division line, and the program ends there; “after” is never printed. The same error occurs in the remainder operation: `5 % 0` also stops with the message “/ by zero”.

With floating-point numbers, however, division by zero causes no error. The floating-point standard (IEEE 754) defines special values for these cases. A positive number divided by zero gives `Infinity`, and a negative one gives `-Infinity`. Meaningless operations such as *0.0 / 0* produce `NaN` (Not a Number). `Math.sqrt(-1)` and ∞ − ∞ also give NaN. These values do not stop the program; they are carried into later operations: ∞ + *5* = ∞, and every arithmetic operation involving NaN is again NaN. An unexpected NaN at the very end of a computation usually shows that there was a division by zero on a much earlier line.

The strangest property of NaN is that it is not equal to itself: `x == x` gives false if x is NaN. That is why a NaN test cannot be done with `x == Double.NaN`; it always returns false. The correct way is the `Double.isNaN(x)` method.

Common mistake

Forgetting that the divisor in a floating-point division can be zero. Integer division shows this error immediately with an exception; floating-point division produces Infinity or NaN and silently continues. If the divisor can be zero, it is checked before the division.

6.4.3 Increment and decrement

Increasing or decreasing a variable by 1 is done so often that Java offers separate operators for it: `++` and `--`. Written as a statement on their own, `i++;` is the same as `i = i + 1;`. A subtle difference appears when the operator is used inside an expression.

Definition 6.9

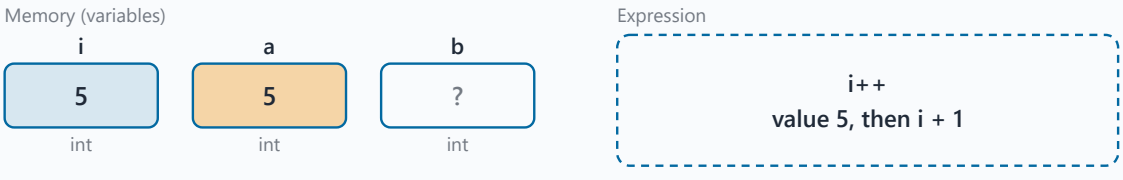
In the **postfix** form (`i++`), the value of the expression is the variable's **old** value; the variable is incremented after this value has been used. In the **prefix** form (`++i`), the variable is incremented first; the value of the expression is the **new** value.

```
int i = 5;
int a = i++;   // use, then increment
int b = ++i;   // increment, then use
i--;
System.out.println(i + " " + a + " " + b);
```

```
6 5 7
```

Line	Statement	Value of the expression	i	a	b
1	`int i = 5;`	–	5	–	–
2	`int a = i++;`	`i++` → 5 (old value)	6	5	–
3	`int b = ++i;`	`++i` → 7 (new value)	7	5	7
4	`i--;`	statement on its own	6	5	7
5	`System.out.println(…);`	output: 6 5 7	6	5	7

Table 6.14: Trace table of the prefix and postfix increment example.



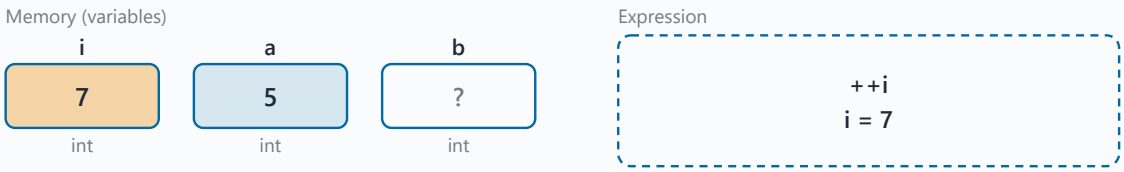


Figure 6.19: The second and third lines. Postfix `i++` : the value of the expression is 5, so a = 5, and then i goes up to 6. Prefix `++i` : first i becomes 7, the value of the expression is the new value 7, and b = 7.

On the second line, `i++` first produces the old value, 5; this value is written to a. The increment also happens on the same line, and i becomes 6. On the third line, `++i` first makes i 7 and then produces this new value; b = 7. On the fourth line, `i--` is a statement on its own. Because its value is not used anywhere, the difference between prefix and postfix does not matter here: if `--i;` had been written, i would also have been 6.

The difference between prefix and postfix invites traps. The line `j = j++;` does not increment j: the right side produces the old value, 5; j briefly becomes 6, and then the assignment writes 5 again. Incrementing the same variable several times in one expression also produces hard-to-read code. The expression `k++ + k++` gives *1 + 2 = 3* for k = 1 and makes k 3. The results are determined by exact rules in Java, but they make every reader stop and think. Using ++ and -- only as a statement on their own keeps the code clear.

6.4.4 Compound assignment and implicit narrowing

The compound assignment operators, `+=` , `-=` , `*=` , `/=` , and `%=` , are not just a short notation. They narrow the result automatically to the type of the variable on the left. The Java language specification defines the statement `x op= y` as `x = (T) (x op y)` , where T is the type of x.

```
byte b = 10;
b += 5;
int i = 5;
i += 3.7;
// b = b + 5;   // won't compile: int → byte
System.out.println(b + " " + i);
```

```
15 8
```

Line	Statement	What the compiler sees	b	i
1	`byte b = 10;`		10	–
2	`b += 5;`	`b = (byte) (b + 5)` : 10 + 5 = 15 (int) → 15 (byte)	15	–
3	`int i = 5;`		15	5
4	`i += 3.7;`	`i = (int) (i + 3.7)` : 5 + 3.7 = 8.7 (double) → 8	15	8

Line	Statement	What the compiler sees	b	i
5	`// b = b + 5;`	comment: disabled because it would not compile	15	8
6	`System.out.println(b + " " + i);`	output: 15 8	15	8

Table 6.15: Trace table of the compound assignment example.

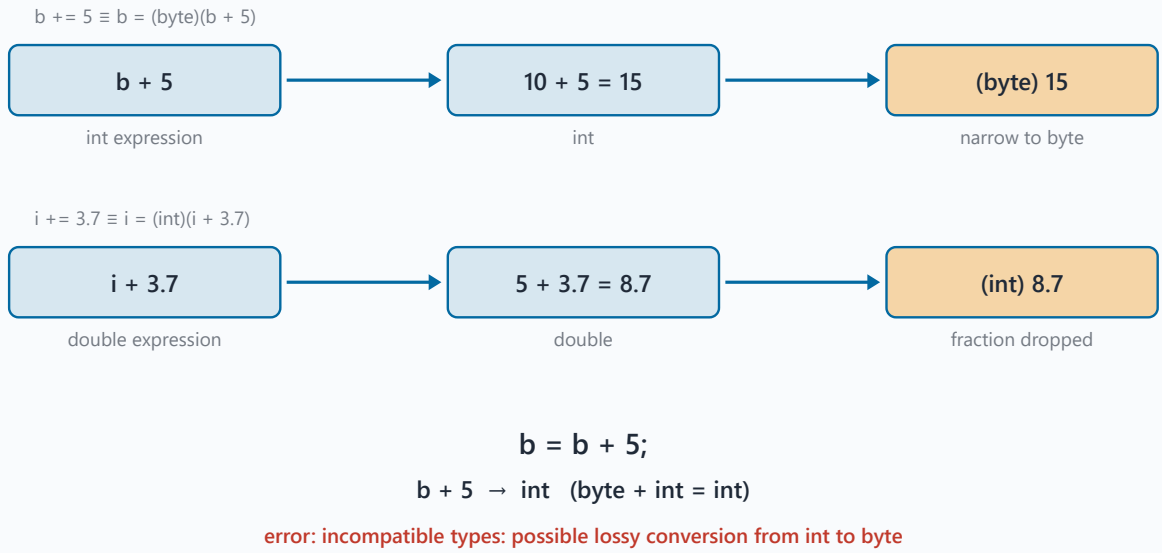


Figure 6.20: The implicit conversion of compound assignment. `b += 5` narrows the int sum to byte. `i += 3.7` narrows the double sum to int and drops the fraction. Writing the same thing as `b = b + 5;` gives a compile error.

The comment on the fifth line is important. In the line `b = b + 5;`, `b + 5` is an int and assigning an int to a byte requires an explicit cast; the line does not compile. `b += 5;`, which does the same job, does compile, because the conversion is hidden in the definition of compound assignment. This convenience also carries a danger: the conversion is silent. `i += 3.7` lowers the result 8.7 to 8 without a warning. When the byte limit is exceeded, overflow also happens silently:

```
int i = 5;
i *= 3 + 7;
System.out.println(i);
int x = 17;
x %= 5;
System.out.println(x);
int y = 20;
y /= 3;
System.out.println(y);
double z = 20;
z /= 3;
System.out.println(z);
byte b = 120;
b += 10;
System.out.println(b);
short s = 1;
s++;
System.out.println(s);
char c = 'a';
c -= 32;
System.out.println(c);
```

```
50
2
6
6.666666666666667
-126
2
A
```

The lines of the output are read in order. In the statement `i *= 3 + 7`, the right side is evaluated completely first: *5 · (3 + 7) = 50*. `x %= 5` turns 17 into the remainder, 2. `y /= 3` is integer division in int: *20 / 3 = 6*; the same operation with the double z gives 6.666666666666667. `b += 10` subtracts 256 to fit 120 + 10 = 130 into a byte: −126. The last two lines show that the `++` and `-=` operators also keep the variable's type: short s becomes 2, and char c becomes 'A' by subtracting 32 from 'a'.

Common mistake

Thinking that the statements `x += y` and `x = x + y` are the same in every case. There are two differences. First, compound assignment does implicit narrowing; `x = x + y` does not compile when the types do not match. Second, in compound assignment the right side is a unit: `x *= a + b` means `x = x * (a + b)`, not `x = x * a + b`.

6.4.5 Comparison and logical operators

Programs make decisions: print pass if the score is greater than 60, warn if the age is below 18. At the base of these decisions are expressions whose result is true or false.

Operator	Meaning	Example
`== !=`	Equal / not equal	`a == b`
`< <= > >=`	Relational comparison	`not >= 60`
`&&`	Both conditions true	`age >= 18 && hasDoc`
`||`	At least one true	`weekend || holiday`
`!`	Logical not	`!active`

Table 6.16: Comparison and logical operators (the table in the slides).

Comparison operators compare two values and always produce a boolean result. Equality is written with two equal signs: `==`. A single equal sign (`=`) is an assignment; confusing the two is one of the most common mistakes. `!=` means "not equal". The mathematical signs ≤ and ≥ are written with two characters, as `<=` and `>=`.

Logical operators combine boolean values. `&&` (and) is true if both conditions are true. `||` (or) is true if at least one is true. `!` (not) takes a single operand and reverses the value. All four possible cases are shown in a truth table:

p	q	p && q	p || q	!p
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Table 6.17: Truth table for && and ||.

```
int age = 20;
boolean hasDoc = true;
System.out.println(age >= 18 && hasDoc);
System.out.println(!hasDoc);
System.out.println(age > 65 || age < 12);
System.out.println(5 > 3 == true);
int score = 75;
System.out.println(score >= 60 && score <= 100);
System.out.println(true ^ true);
```

```
true
false
false
true
true
false
```

With age 20 and a document present, `age >= 18 && hasDoc` is true. The expression `age > 65 || age < 12` is false because both conditions are false. Whether a value is within a range is written by combining two comparisons with `&&`: `score >= 60 && score <= 100`. The `^` in the last line is the "either one or the other" (exclusive or) operator; it is true if the two operands differ.

The notation *0 < x < 10* customary in mathematics does not work in Java:

```
error: bad operand types for binary operator '<'
System.out.println(0 < x < 10);
                       ^
  first type:  boolean
  second type: int
1 error
```

`0 < x` is evaluated first and produces a boolean value. Then the program tries to compare this boolean value with 10; `<` is not defined between a boolean and an int. The correct form is `0 < x && x < 10`.

6.4.6 Operator precedence and the expression tree

Is the value of the expression `2 + 3 * 4` 20 or 14? In mathematics it is known that multiplication is done before addition, and the answer is 14. Java follows the same rule and extends it to all operators.

Definition 6.10

Operator precedence is the set of rules that determines in which order different operators in an expression are applied. Operators of the same precedence are applied from left to right; assignment operators are applied from right to left.

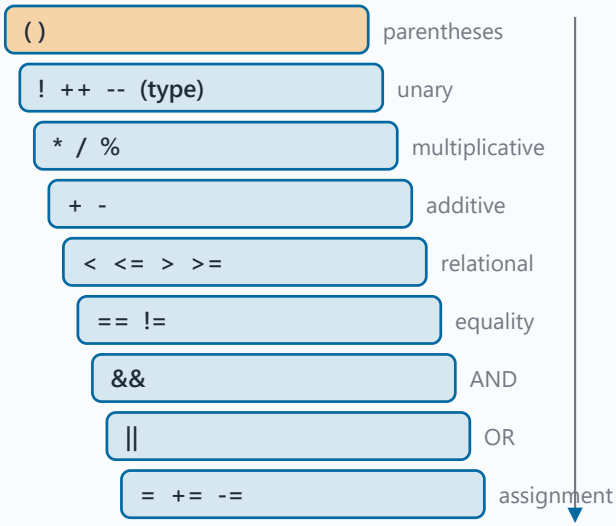


Figure 6.21: The operator precedence ladder. The upper step is applied first: parentheses, unary operators (`!`, `++`, `--`, type cast), multiplicative, additive, relational comparison, equality, `&&`, `||`, and assignment last.

The ladder corresponds to the four rules in the slides. The inside of parentheses is always evaluated first; the unary `!` and the type cast are also applied early. Multiplication, division, and remainder come before addition and subtraction. Comparisons are evaluated after arithmetic, and `&&` and `||` after comparisons; `&&` comes before `||`. Operators at the same level are applied from left to right. That is why `10 - 4 - 3` is *(10 − 4) − 3 = 3*, not *10 − (4 − 3) = 9*. `100 / 10 / 5` is likewise *(100 / 10) / 5 = 2*. Assignment, however, works from right to left: `a = b = 4;` first writes 4 into b and then writes this value into a.

When the compiler reads an expression, it turns it into an **expression tree** according to precedence. Literals and variables are at the leaves of the tree and operators are at the inner nodes. The operations at the bottom are evaluated first, and the operation at the root is evaluated last. The expression from the slides shows this process step by step:

```
2 + 3 * 4 > 10 && 7 % 2 == 1
```

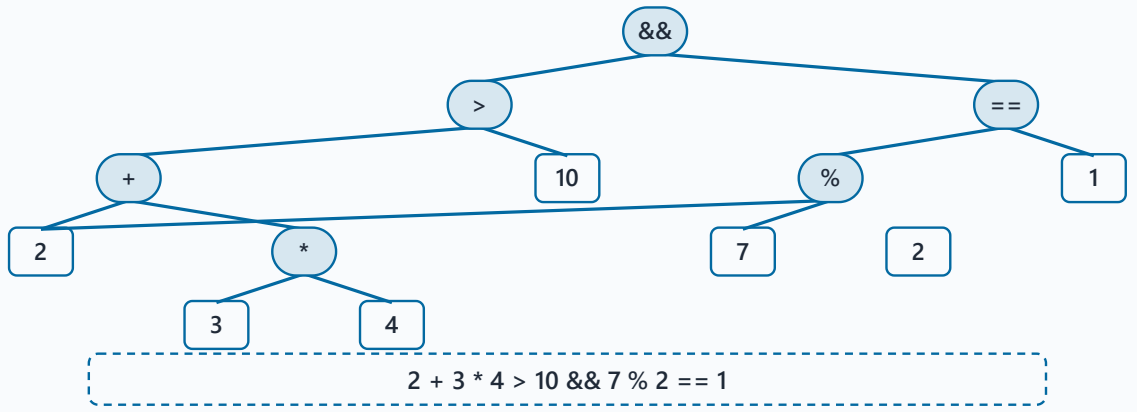
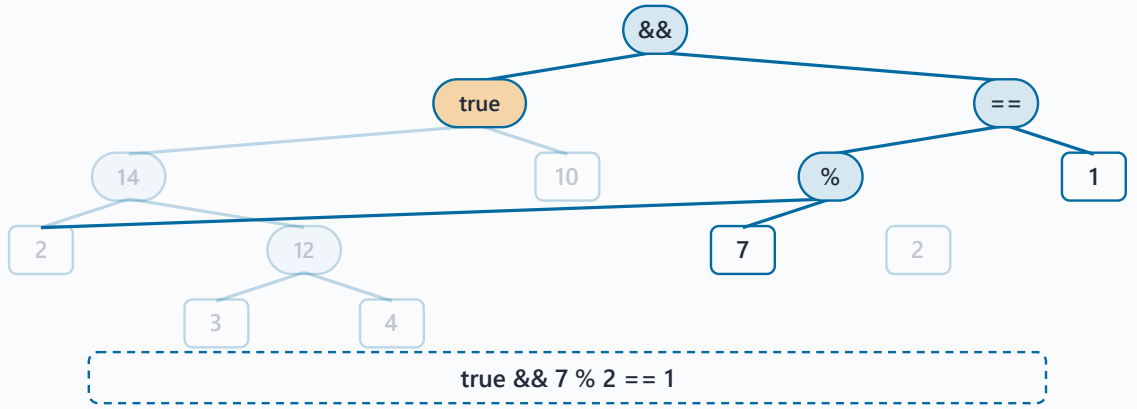


Figure 6.22: The tree of the expression `2 + 3 * 4 > 10 && 7 % 2 == 1`. At the root is the lowest-precedence operator, `&&`; its two subtrees are the two comparisons.

Step	Operation computed	New form of the expression
0	–	`2 + 3 * 4 > 10 && 7 % 2 == 1`
1	`3 * 4 = 12` (* before +)	`2 + 12 > 10 && 7 % 2 == 1`
2	`2 + 12 = 14`	`14 > 10 && 7 % 2 == 1`
3	`14 > 10` → true	`true && 7 % 2 == 1`
4	`7 % 2 = 1`	`true && 1 == 1`
5	`1 == 1` → true	`true && true`
6	`true && true` → true	`true`

Table 6.18: Step-by-step simplification of the expression. At each step the deepest operation is evaluated.



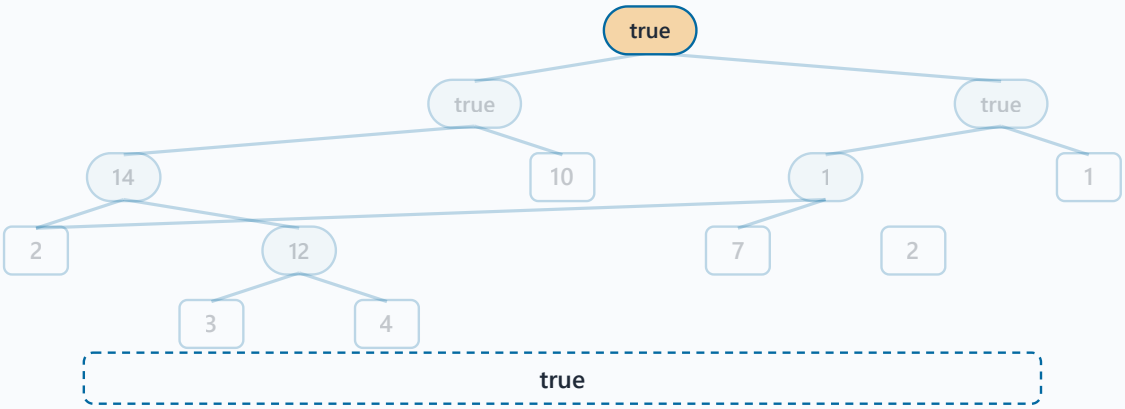


Figure 6.23: Steps 3 and 6 of the tree. The left subtree has been reduced to the value true. Because the left side is true, `&&` also evaluates the right side; in the last step the root becomes true.

A change of type happens at the third step: the comparison `14 > 10` takes two ints but produces a boolean. In the upper levels of the tree, logical values now circulate instead of numbers. Both operands of the `&&` operator must be boolean. Because the left side is true, the result is not yet determined; the right side is evaluated too. If the left side had been false, the right side would never have been looked at.

```
System.out.println(2 + 3 * 4);
System.out.println((2 + 3) * 4);
System.out.println(10 - 4 - 3);
System.out.println(100 / 10 / 5);
System.out.println(2 + 3 * 4 > 10 && 7 % 2 == 1);
System.out.println(10 - 4 / 2 * 3 + 7 % 4);
System.out.println(-3 * 2 + 8 / 3 % 2);
int a, b;
a = b = 4;
System.out.println(a + " " + b);
boolean p = true, q = false, r = true;
System.out.println(p || q && !r);
System.out.println((p || q) && !r);
```

```
14
20
3
2
true
7
-6
4 4
true
false
```

The last two lines show the precedence of the logical operators. In the expression `p || q && !r`, first `!r`, then `&&`, and finally `||` is applied: *true || (false && false) = true*. When the order is changed with parentheses, the result changes too: *(true || false) && false = false*. In the expression `-3 * 2 + 8 / 3 % 2`, the unary minus comes first, then *, /, and % are applied from left to right: *−6 + (8 / 3) % 2 = −6 + 2 % 2 = −6 + 0 = −6*.

There is no need to memorize all the precedence rules. Using parentheses wherever the reader might be left with a question mark both prevents errors and makes the intent visible. Writing `(a + b) / 2` is easier than working out what `a + b / 2` means.

6.4.7 A mixed expression: precedence and type together

In real expressions the precedence and type rules work together. The example from the slides shows how the two interact:

```
int a = 7, b = 2;
double r = a / b * 2.0 + a % b;
System.out.println(r);
```

```
7.0
```

Step	Operation	Types	New form of the expression
1	`a / b` = 7 / 2 = 3	int / int → int (integer division)	`3 * 2.0 + a % b`
2	`3 * 2.0` = 6.0	int * double → double	`6.0 + a % b`
3	`a % b` = 7 % 2 = 1	int % int → int	`6.0 + 1`
4	`6.0 + 1` = 7.0	double + int → double	`7.0`

Table 6.19: Step-by-step computation of the expression `a / b * 2.0 + a % b` for a = 7, b = 2.

3 * 2.0 + a % b

a / b = 7 / 2 = 3 (int / int → integer division)

6.0 + a % b

3 * 2.0 = 6.0 (int → double, result double)

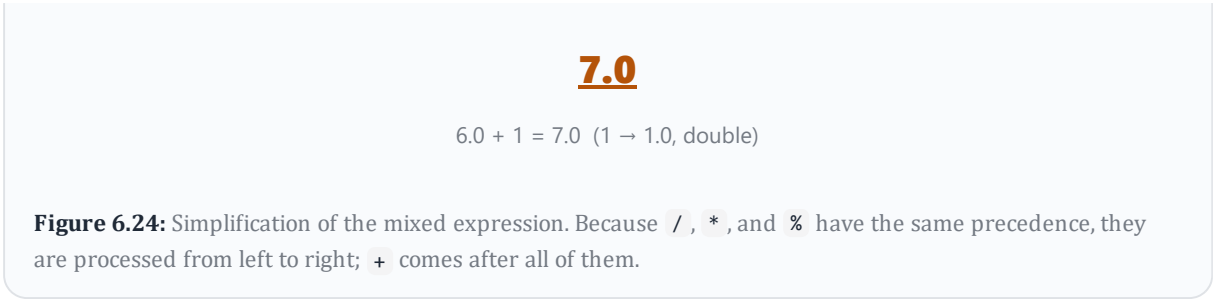


Figure 6.24: Simplification of the mixed expression. Because `/`, `*`, and `%` have the same precedence, they are processed from left to right; `+` comes after all of them.

/, *, and % are at the same precedence level and are processed from left to right; + comes after all of them. The first operation is `a / b`, and because both are ints it gives 3, not 3.5. The presence of a double such as 2.0 in the expression does not affect this first operation; 2.0 comes into play only in the second operation. The result is 7.0. If `a / 2.0` had been written instead of `a / b`, the first operation would give 3.5 and the result would be *3.5 · 2.0 + 1 = 8.0*. Getting two different answers with the same numbers shows that the result is determined by the order of the operators and the operand types together.

6.4.8 Short-circuit evaluation

The `&&` and `||` operators have a property that speeds up their work and makes programs safe.

Definition 6.11

Short-circuit evaluation is the practice of not evaluating the right operand at all when the result of a logical expression is already determined by the left operand. For `&&`, if the left side is false the result is false; for `||`, if the left side is true the result is true. In these cases the right side is not executed.

```
int a = 10, b = 0;
if (b != 0 && a / b > 2) {
    System.out.println("Large ratio");
}
b = 2;
if (b != 0 && a / b > 2) {
    System.out.println("Large ratio");
}
```

```
Large ratio
```

The program tests the same condition with two different values of b. The left side of the condition is `b != 0` and the right side is `a / b > 2`. (An if statement runs the lines inside the braces if the condition is true and skips them otherwise.)

b	Left: `b != 0`	Right: `a / b > 2`	Condition	Output
0	0 != 0 → false	not evaluated	false	none (block skipped)
2	2 != 0 → true	10 / 2 = 5; 5 > 2 → true	true	Large ratio

Table 6.20: Trace table of the short-circuit example.

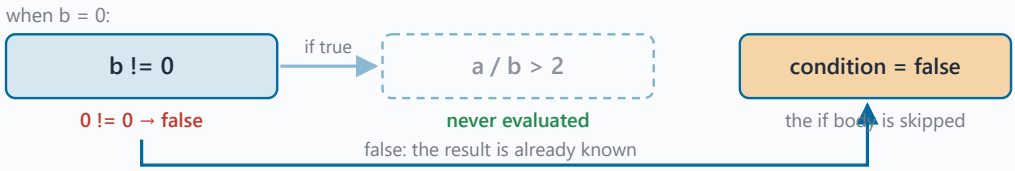


Figure 6.25: Short-circuit with b = 0. Because the left condition is false, the result of the `&&` expression is determined; `a / b > 2` is never evaluated and no division-by-zero error occurs.

If the right side had been executed with b = 0, `a / b` would have divided an integer by zero and the program would have crashed with an `ArithmeticException`. Short-circuiting prevents this: when the left side is false, the result is false whatever happens, and Java does not look at the right side. That is why the order of conditions matters. If the conditions are written in reverse, the protection disappears:

```
int a = 10, b = 0;
if (a / b > 2 && b != 0) {
    System.out.println("Large ratio");
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Here `a / b` is evaluated first, and the program stops before the `b != 0` check gets its turn. The guarding condition is always written on the left and the guarded operation on the right. For `||` the same idea works in reverse: in the expression `b == 0 || a / b > 2`, if b = 0 the left side is true and the division is not performed.

```
int a = 10, b = 0;
if (b == 0 || a / b > 2) {
    System.out.println("b is zero or ratio large");
}
int count = 0;
boolean s1 = false && (++count > 0);
boolean s2 = true || (++count > 0);
boolean s3 = false & (++count > 0);
System.out.println(count + " " + s1 + " " + s2 + " " + s3);
```

```
b is zero or ratio large
1 false true false
```

The counter in the second part counts which right sides actually ran. In the expressions `false && (++count > 0)` and `true || (++count > 0)`, the right side does not run because of short-circuiting. The single `&`, however, is the “and” operator that does not short-circuit; it always evaluates both sides. The counter being 1 shows that the increment happened only in this third expression. In everyday programming, `&&` and `||` are always used for conditions.

6.4.9 String concatenation

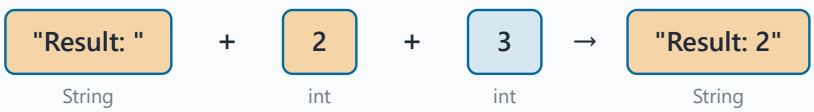
The `+` operator also has two faces. If the two operands are numbers, it adds. If at least one of the operands is a string (String), it performs **string concatenation**: the other operand is converted to text and the two texts are joined end to end. The types of the operands determine the operator's behavior.

```
System.out.println(2 + 3);       // 5
System.out.println("2" + "3");   // 23
System.out.println("Result: " + 2 + 3); // Result: 23
System.out.println("Result: " + (2 + 3)); // Result: 5
```

Two rules are enough to explain the results of the four lines. First, `+` works from left to right. Second, at each step the two operands at that step are examined: if both are numbers, addition is done; if one is a string, concatenation is done.

Expression	Step 1	Step 2	Output
`2 + 3`	int + int → addition: 5	–	5
`"2" + "3"`	String + String → concatenation: "23"	–	23
`"Result: " + 2 + 3`	String + int → "Result: 2"	String + int → "Result: 23"	Result: 23
`"Result: " + (2 + 3)`	parentheses: int + int → 5	String + int → "Result: 5"	Result: 5

Table 6.21: Steps of the four `println` lines.



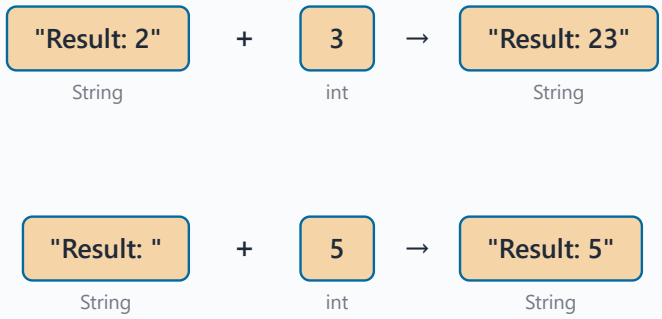


Figure 6.26: In the expression `"Result: " + 2 + 3`, first `"Result: " + 2` is concatenated and becomes "Result: 2"; then 3 is also appended to the text. In the form with parentheses, first *2 + 3 = 5* is added, and then "Result: 5" is formed.

The third line is the most surprising. The reader expects 2 and 3 to be added, but Java goes from left to right: in the first step the string “Result: ” is joined with 2. The result of this step is a string; in the second step the string is joined with 3. 2 + 3 is never added. Parentheses make the addition happen first. The rule also holds at the start of an expression:

```
System.out.println(2 + 3);
System.out.println("2" + "3");
System.out.println("Result: " + 2 + 3);
System.out.println("Result: " + (2 + 3));
System.out.println(2 + 3 + " apples");
System.out.println("Result: " + 2 * 3);
System.out.println("Result: " + 7 / 2);
System.out.println('A' + 1 + "B");
System.out.println("A" + 'B' + 1);
System.out.println(1 + 'A');
System.out.println("x" + 1.5 + true);
String s = "Result: ";
s += 5;
System.out.println(s);
```

```
5
23
Result: 23
Result: 5
5 apples
Result: 6
Result: 3
66B
AB1
66
x1.5true
Result: 5
```

- `2 + 3 + " apples"`: the two operands on the left are numbers and are added first; the result is "5 apples".
- `"Result: " + 2 * 3` and `"Result: " + 7 / 2`: * and / have higher precedence than +. The numeric operation is done first (6 and 3), then the concatenation.
- `'A' + 1 + "B"`: in the first step a char and an int are added (*65 + 1 = 66*), then "B" is appended: "66B". `"A" + 'B' + 1`, however, starts with a string from the beginning and so gives "AB1".
- `"x" + 1.5 + true`: double and boolean values are also converted to text: "x1.5true".
- `s += 5`: compound assignment also works with strings and appends to the end of the text.

Another warning about strings concerns comparison. Whether two strings have the same content is tested not with `==` but with the `equals` method. For reference types such as String, `==` checks whether two variables point to the same object, not the content:

```
String s1 = "Ada";
String s2 = new String("Ada");
System.out.println(s1 == s2);
System.out.println(s1.equals(s2));
```

```
false
true
```

`s1` and `s2` carry the same text "Ada" but are two separate objects in memory. `==` gives false and `equals` gives true.

6.5 Input and output

The programs so far have always worked with the same numbers; the values were written into the code. A program that converts temperatures or computes a grade average, however, must work with different values every time it runs. These values are taken from the user, usually from the keyboard. In Java the easiest way to read from the keyboard is the `Scanner` class. The `printf` method is used to write results neatly aligned and with a given number of decimal places.

6.5.1 Input with Scanner

```
Scanner in = new Scanner(System.in);
System.out.print("Age: ");
int age = in.nextInt();
in.nextLine(); // consume the newline
System.out.print("Name: ");
String name = in.nextLine();
```

Scanner is in Java's `java.util` package. Before it is used, the line `import java.util.Scanner;` is written at the very top of the file, before the class definition. This line tells the compiler where the name Scanner comes from; if it is not written, a “cannot find symbol” error occurs.

1. `Scanner in = new Scanner(System.in);` : `new` creates a Scanner object and connects it to standard input, that is, to the keyboard (`System.in`). `in` is the variable that holds a reference to this object; its name can be chosen freely.
2. `System.out.print("Age: ");` : writes a **prompt** that tells the user what to enter. Because `print` is used instead of `println` , no line break is added; the user types on the same line.
3. `int age = in.nextInt();` : the program stops and waits for the user to type an integer and press Enter. The number read is written into the `age` variable.
4. `in.nextLine();` : the Enter character left at the end of the line is consumed. `nextInt()` does not read the Enter after the number; this call takes it out of the buffer.
5. `String name = in.nextLine();` : the whole line the user typed, spaces included, is read.

Scanner's read methods are chosen according to the expected type: `nextInt()` reads an int, `nextLong()` a long, `nextDouble()` a double, `nextBoolean()` a boolean. `next()` reads a single word up to whitespace, and `nextLine()` reads everything up to the end of the line. The methods that read numbers and words treat whitespace and line breaks in the input as separators and skip them. Whether the user types the numbers 20, 30, and 40 on the same line separated by spaces or on separate lines, three `nextInt()` calls read all three.

Here is how the program looks on the screen together with what the user types (the user types 20 and Ada Yılmaz):

```
Age: 20
Name: Ada Yılmaz
```

6.5.2 The input buffer, step by step

What the user types reaches Scanner not directly but through a **buffer**. The buffer is a queue in which the characters coming from the keyboard wait until they are read. When the user presses Enter, the line typed enters the buffer together with the line break character at its end. The line break is written `\n` in Java and shown with the symbol ⏎ in the figures. Each read method of Scanner takes certain characters from the buffer and leaves the rest for later reads.

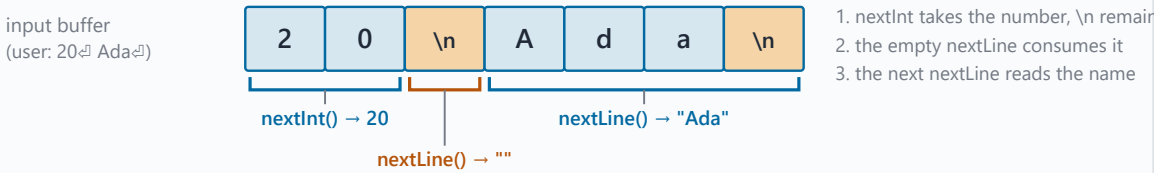


Figure 6.27: The input buffer: the user first types 20⏎, then Ada⏎. `nextInt()` reads 20 and leaves the line break in the buffer. The empty `nextLine()` consumes this line break. The next `nextLine()` reads the name.

Statement	What happens?	Read	Buffer
`new Scanner(System.in)`	Scanner is connected to the keyboard	–	empty
`print("Age: ")`	the prompt is printed; the cursor stays on the same line	–	empty
`nextInt()`	buffer empty: waits. The user types 20⏎	20	⏎
`nextLine()`	reads up to ⏎, consumes ⏎	"" (empty string)	empty
`print("Name: ")`	the prompt is printed	–	empty
`nextLine()`	buffer empty: waits. The user types Ada Yılmaz⏎	"Ada Yılmaz"	empty

Table 6.22: Trace table of the Scanner example. The Buffer column shows the characters waiting to be read after the statement has run.





Figure 6.28: Three moments of the buffer. `nextInt` reads only 20; the read position stops in front of ⏎. `nextLine` reads up to ⏎ and returns an empty string. The last `nextLine` reads the whole line: "Ada Yılmaz". The characters already read are shown faded.

The third row of the table explains everything. `nextInt()` reads the digits of the number and stops at the first character that is not a digit. This character is the Enter the user pressed, and it stays in the buffer. `nextLine()`, however, reads "the rest of the line, up to the line break". The line break character is right at the read position; that is why the first `nextLine()` returns an empty string without waiting at all and consumes the line break. The value returned by this call is not assigned anywhere; the only purpose of the call is to clear the buffer. The last `nextLine()` meets an empty buffer and waits for the user. The "Ada Yılmaz" the user types is read with its space; the line break is consumed but not added to the returned string.

6.5.3 When the end of line is not consumed

What happens if the line that clears the buffer is forgotten? The second program in the slides shows this case:

```
Scanner in = new Scanner(System.in);
System.out.print("Age: ");
int age = in.nextInt();
System.out.print("Name: ");
String name = in.nextLine();
System.out.println("[" + name + "]");
```

When the user types 20 and presses Enter, this appears on the screen and the program ends:

```
Age: 20
Name: []
```



Figure 6.29: When the line break is not consumed: the ⏎ left over from `nextInt` is in the buffer. `nextLine` reads up to this ⏎ without waiting and returns an empty string; the user cannot be asked for the name.

The program seems to end without asking for the name. In reality the “Name: ” prompt has been printed, `nextLine()` has been called, and it has found the line break waiting in the buffer and returned at once. The `name` variable is an empty string; there is nothing between the square brackets. The user gets no chance to type a second thing. The solution is to call an extra `in.nextLine();` after `nextInt()`, as in the first program in the slides. Another solution is to read the numbers as lines too and convert them: `int age = Integer.parseInt(in.nextLine());`.

The same effect is seen when the user types the number and the name on the same line. If the input is “20 Ada Yılmaz⏎”, `nextInt()` reads 20 and the next `nextLine()` reads the rest of the line: “ Ada Yılmaz”. The leading space is part of the string. The difference between `next()` and `nextLine()` shows up similarly: for the input “Ada Yılmaz⏎”, `next()` reads only “Ada”, and the `nextLine()` that follows reads “ Yılmaz”.

Common mistake

Trying to read a line with `nextLine()` immediately after a call to `nextInt()`, `nextDouble()`, or `next()`. The first `nextLine()` reads the line break left over from the previous line and returns an empty string. If a line is to be read after reading a number, an `in.nextLine();` that clears the buffer is put in between.

6.5.4 Input validation

It cannot be assumed that the user will always enter the expected value. In a program that asks for an age, “twenty” may be typed, and −5 or 200 may be entered by mistake. When `nextInt()` meets a word that is not a number where it expects a number, it stops the program:

```
Age: twenty
Exception in thread "main" java.util.InputMismatchException
```

To make the program robust against such input, **input validation** is performed. Validation has two layers. First the **type** is checked: is what was entered really an integer? Then the **range and business rule** are checked: is the integer a reasonable age?

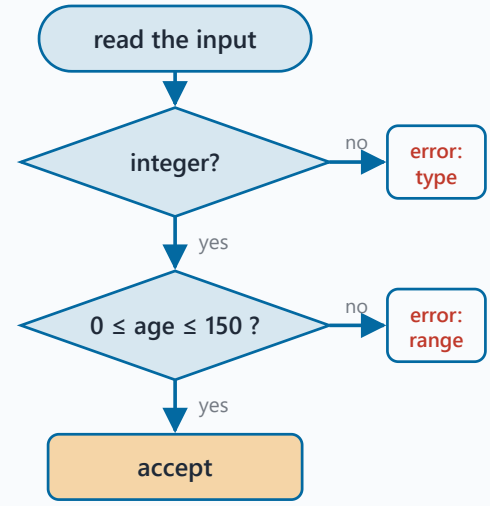


Figure 6.30: The input validation flow. First the type of the input is checked, then its range. A value that passes both checks is accepted; for one that fails, an error message is shown and the value is asked for again.

Scanner offers methods such as `hasNextInt()` for type checking. This method looks at the input without reading it: it returns true if the next piece is an integer and false otherwise. The following program does both checks with `if` statements. The structure `if (condition) { … } else { … }` runs the first block if the condition is true and the block after `else` if it is false.

```
Scanner in = new Scanner(System.in);
System.out.print("Age: ");
if (in.hasNextInt()) {
    int age = in.nextInt();
    if (age >= 0 && age <= 150) {
        System.out.println("Accepted: " + age);
    } else {
        System.out.println("Error: age must be between 0 and 150.");
    }
} else {
    System.out.println("Error: integer expected (example: 20).");
}
```

When the program is run with three different inputs, it gives these outputs:

```
Age: twenty
Error: integer expected (example: 20).

Age: 200
Error: age must be between 0 and 150.

Age: 20
Accepted: 20
```

When “twenty” is entered, `hasNextInt()` returns false; because `nextInt()` is never called, the program does not crash and writes a clear message. When 200 is entered, the type check passes but the range check fails. 20 passes both checks. Note that the error message states the expected form explicitly: writing “integer expected (example: 20)” instead of “Invalid input” shows the user what to do. In a real program the value is asked for again after a wrong input; this repetition is done with a loop.

6.5.5 Formatted output: printf

`println` writes values as they are: the value 87.456 appears as 87.456 and 2.0/3 as 0.6666666666666666. In a grade table, however, the averages should be written with two decimal places and the columns aligned. The `printf` method, which produces **formatted output**, is used for this job.

```
String name = "Ada";
double avg = 87.456;
System.out.printf("%-10s %6.2f%n", name, avg);
```

```
Ada         87.46
```

The first argument of `printf` is a **format string**. Ordinary characters in the format string are written as they are. The parts that start with % are **format specifiers**: each describes how the next argument is to be written and consumes the arguments in order.

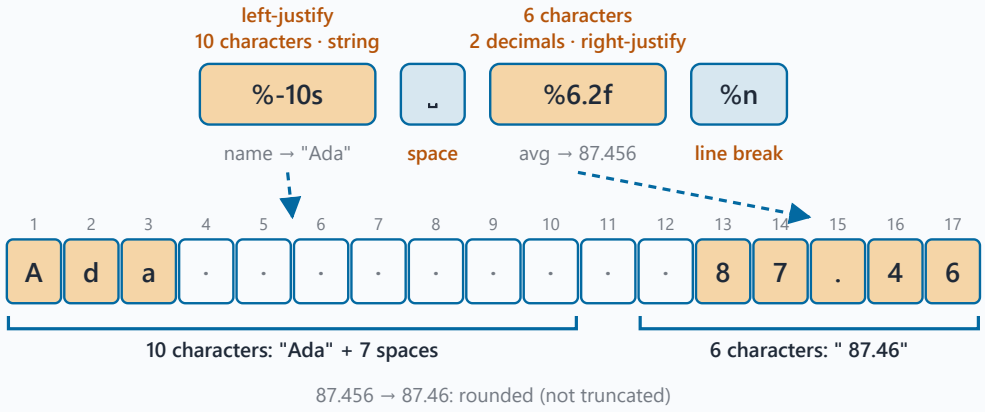


Figure 6.31: Parsing the format string `%-10s %6.2f%n`. `%-10s` writes "Ada" left-justified in a field of 10 characters (3 letters + 7 spaces). Then comes a space. `%6.2f` rounds 87.456 to two decimal places and right-justifies it in a field of 6 characters: " 87.46".

A format specifier starts with the % sign and ends with a **conversion character**. The three parts in between are optional. **Flags** determine alignment and padding: `-` left-justifies, `0` pads with zeros instead of spaces, `,` adds a thousands separator. The **width** says how many characters the field will have at least. The **precision** is written after a period and gives the number of decimal places. The most frequently used conversion characters are these:

Specifier	Meaning	Example	Output
`%d`	integer	`printf("[%d]", 42)`	`[42]`
`%5d`	5 characters, right-justified	`printf("[%5d]", 42)`	`[ 42]`
`%-5d`	5 characters, left-justified	`printf("[%-5d]", 42)`	`[42 ]`
`%05d`	5 characters, zero-padded on the left	`printf("[%05d]", 42)`	`[00042]`
`%,d`	with a thousands separator	`printf("[%,d]", 1234567)`	`[1,234,567]`
`%f`	floating-point, 6 digits	`printf("[%f]", Math.PI)`	`[3.141593]`
`%.3f`	3 decimal places	`printf("[%.3f]", 2.0005)`	`[2.001]`
`%e`	scientific notation	`printf("[%e]", 12345.678)`	`[1.234568e+04]`
`%s`	string (every value is converted to text)	`printf("[%s]", 3.5)`	`[3.5]`
`%c`	character	`printf("[%c]", 'Z')`	`[Z]`
`%b`	boolean	`printf("[%b]", false)`	`[false]`
`%n`	line break	`printf("a%nb")`	a and b on separate lines
`%%`	the % sign itself	`printf("%%%d off", 20)`	`%20 off`

Table 6.23: Frequently used format specifiers. The outputs are given in square brackets so that the field width is visible (JDK 21, Locale.US).

The examples in the table are taken from the following program. The program also shows two more details. The width is a lower bound: the specifier `%3s` does not truncate an 8-letter name but writes it as it is; `%2d` does not truncate 12345 either. Also, the `String.format` method does the same

formatting but returns the result as a string instead of printing it.

```
String name = "Ada";
double avg = 87.456;
System.out.printf("%-10s %6.2f%n", name, avg);
System.out.printf("[%-10s][%6.2f]%n", name, avg);
System.out.printf("%-8s|%6.2f|%05d%n", "Ada", 3.14159, 42);
System.out.printf("%-6s|%7.3f|%04d%n", "Can", 2.71828, 7);
System.out.printf("[%5d][%-5d][%05d][%d]%n", 42, 42, 42, 42);
System.out.printf("[%.0f][%.1f][%.3f][%f]%n", 2.5, 2.45, 2.0005, Math.PI);
System.out.printf("[%e][%.2e]%n", 12345.678, 0.000123);
System.out.printf("[%,d][%,.2f]%n", 1234567, 1234567.891);
System.out.printf("[%s][%s][%s][%b][%c]%n", 42, 3.5, true, false, 'Z');
System.out.printf("%%%d off%n", 20);
System.out.printf("[%3s][%2d]%n", "LongName", 12345);
System.out.printf("[%.1f][%.1f][%.1f][%.1f]%n", 0.25, 0.35, 0.05, 2.25);
String s = String.format("%d:%02d:%02d", 1, 2, 5);
System.out.println(s);
```

```
Ada         87.46
[Ada       ][ 87.46]
Ada     |  3.14|00042
Can   |  2.718|0007
[   42][42   ][00042][42]
[3][2.5][2.001][3.141593]
[1.234568e+04][1.23e-04]
[1,234,567][1,234,567.89]
[42][3.5][true][false][Z]
%20 off
[LongName][12345]
[0.3][0.4][0.1][2.3]
1:02:05
```

The values 0.25, 0.35, 0.05, and 2.25 written with `%.1f` are all rounded up: 0.3, 0.4, 0.1, 2.3. When rounding, printf looks at the decimal representation of the number and rounds halves up.

The second example in the slides uses three specifiers on one line:

```
System.out.printf("%-8s|%6.2f|%05d%n",
                  "Ada", 3.14159, 42);
```

```
Ada     |  3.14|00042
```

Part	Argument	Rule	Produced
`%-8s`	"Ada"	8 characters, `-` left-justifies: 3 letters + 5 spaces	`Ada␣␣␣␣␣`
`|`	–	plain character, written as it is	`|`
`%6.2f`	3.14159	round to 2 digits (3.14), right-justify in 6 characters: 2 spaces	`␣␣3.14`
`|`	–	plain character	`|`
`%05d`	42	5 characters, the `0` flag pads with zeros on the left	`00042`
`%n`	–	line break	⏎

Table 6.24: Parsing of the format string `"%-8s|%6.2f|%05d%n"` from left to right.

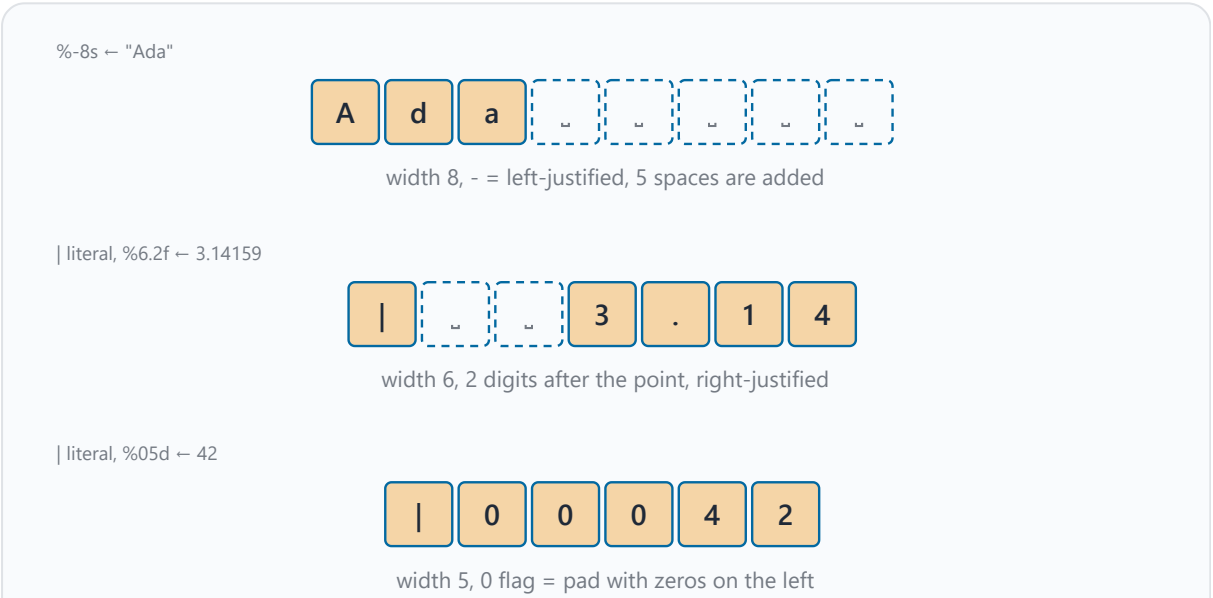


Figure 6.32: The characters produced by three format specifiers. The ␣ symbol shows a space. Each % specifier consumes one argument; the vertical bars are plain text.

The number of specifiers must match the number and types of the arguments. A mismatch is noticed not at compile time but while the program runs. The call `printf("%d%n", 3.5)` throws `IllegalFormatConversionException`, because %d expects an integer and 3.5 is a double. The call `printf("%s and %s%n", "Ada")` fills the first specifier, finds no argument for the second, and stops with `MissingFormatArgumentException`.

A last detail of formatted output is the **locale**. printf chooses the decimal separator and the thousands separator according to the computer's language and region setting. On a computer with a Turkish locale, the same program gives different output:

```
String name = "Ada";
double avg = 87.456;
System.out.printf("%-10s %6.2f%n", name, avg);
System.out.printf("%-8s|%6.2f|%05d%n", "Ada", 3.14159, 42);
System.out.printf("[%,d][%,.2f]%n", 1234567, 1234567.891);
System.out.printf(Locale.US, "%6.2f%n", avg);
System.out.println(avg);
```

```
Ada         87,46
Ada     |  3,14|00042
[1.234.567][1.234.567,89]
 87.46
87.456
```

With the Turkish setting, the decimal separator is a comma and the thousands separator is a period: 87,46 and 1.234.567,89. If `Locale.US` is given as the first argument, the setting is overridden for that call and a period is used. `println`, however, is not affected by the locale; it always uses a period. In class and in the slides, outputs are given in the Locale.US form.

The locale also affects Scanner, and this effect is more annoying. With a Turkish locale, `nextDouble()` expects the decimal number to be written with a comma. If the user types “37.5”, an `InputMismatchException` is thrown; if the user types “37,5”, 37.5 is read. With the Locale.US setting, it is exactly the reverse. If you want the program to behave the same on every computer, the setting is fixed after the Scanner is created: `in.useLocale(Locale.US);`. After this line, the input “37.5” is read on a computer with a Turkish locale as well.

Common mistake

Getting an `InputMismatchException` when a decimal number is entered in a homework program and thinking the code is wrong. The code may be correct; the problem is that the computer, under a Turkish locale, expects a comma instead of a period. Either the input is written with a comma or `useLocale(Locale.US)` is applied to the Scanner. (The Locale class is also in the `java.util` package.)

6.6 Examples

The small programs in this section use all the tools of the chapter together: variables, types, conversions, operators, Scanner, and printf. In each program, the code is given first, then the step-by-step computation, and finally the actual output.

6.6.1 The Math class

For operations such as square root, exponentiation, and absolute value, Java's built-in `Math` class is used. Math is in the `java.lang` package. Because this package is included in every Java program automatically, no `import` line is needed. The methods of Math are called by the class name, without creating an object: `Math.sqrt(16)`.

Expression	Result	Explanation
`Math.PI`	3.141592653589793	the constant π
`Math.sqrt(16)`	4.0	square root (double)
`Math.pow(2, 10)`	1024.0	exponentiation (double)
`Math.abs(-7)`	7	absolute value
`Math.max(3, 8)`	8	the larger (min: the smaller)
`Math.round(2.5)`	3	nearest integer (long)
`Math.round(-2.5)`	-2	a half value is rounded up

Table 6.25: Frequently used Math members (the table in the slides; the values were verified with JDK 21).

`sqrt` and `pow` always return a double; even the square root of the perfect square 16 is 4.0, not 4. `abs`, `max`, and `min`, however, keep the type of the argument: `Math.abs(-7)` gives the int 7 and `Math.abs(-7.5)` gives the double 7.5. If one of the arguments is a double, the result is a double too: `Math.max(3, 8.0)` gives 8.0.



Figure 6.33: Math.round on the number line. Exact half values always round toward positive infinity: 2.5 → 3, −2.5 → −2.

`Math.round` finds the nearest integer. If the number is exactly halfway between two integers (like 2.5), it rounds up, that is, toward positive infinity. That is why 2.5 → 3 but −2.5 → −2; −2 is greater than −3. The rule taught in school, “round a half away from zero”, gives −3 for −2.5; Java's rule is different. Other methods of the Math class can be seen in the following program:

```
System.out.println(Math.PI);
System.out.println(Math.E);
System.out.println(Math.sqrt(16));
System.out.println(Math.pow(2, 10));
System.out.println(Math.abs(-7) + " " + Math.abs(-7.5));
System.out.println(Math.max(3, 8) + " " + Math.min(3, 8));
System.out.println(Math.round(2.5) + " " + Math.round(-2.5) + " " + Math.round(2.4) + " " +
Math.round(-2.6));
System.out.println(Math.floor(2.7) + " " + Math.ceil(2.1) + " " + Math.floor(-2.7) + " " +
Math.ceil(-2.1));
System.out.println(Math.pow(2, 0.5) + " " + Math.sqrt(2));
System.out.println(Math.pow(2, 31) + " " + (int) Math.pow(2, 31));
System.out.println(Math.max(3, 8.0));
System.out.println(Math.sqrt(2) * Math.sqrt(2));
System.out.println(Math.hypot(3, 4));
```

```
3.141592653589793
2.718281828459045
4.0
1024.0
7 7.5
8 3
3 -2 2 -3
2.0 3.0 -3.0 -2.0
1.4142135623730951 1.4142135623730951
2.147483648E9 2147483647
8.0
2.0000000000000004
5.0
```

`Math.floor` rounds a number down (toward negative infinity) and `Math.ceil` rounds it up (toward positive infinity); both give the result as a double: *floor(−2.7) = −3.0*, *ceil(−2.1) = −2.0*. Remembering that the `(int)` cast truncates toward zero, one sees that for negative numbers floor and (int) give different results. `Math.pow(2, 31)` is 2.147483648E9; when this value is converted to int it clamps at the limit and becomes 2147483647. $\sqrt{2} \cdot \sqrt{2}$ is not exactly 2 but 2.0000000000000004, because of floating-point error. `Math.hypot(3, 4)` gives the hypotenuse of a right triangle: $\sqrt{(9 + 16)} = 5$.

6.6.2 Swapping two variables

Exchanging the values of two variables, that is, **swapping** them, is a basic step of many algorithms such as sorting. The first solution that comes to mind is wrong:

```
int a = 5, b = 8;
a = b;                // wrong attempt
b = a;
System.out.println(a + " " + b);
a = 5; b = 8;         // start over
int temp = a;         // the right way
a = b;
b = temp;
System.out.println(a + " " + b);
```

```
8 8
8 5
```

Statement	a	b	temp	Explanation
`int a = 5, b = 8;`	5	8	–	goal: a = 8, b = 5
`a = b;`	8	8	–	the old value of a, 5, was erased
`b = a;`	8	8	–	5 is no longer in any box
`println`	8	8	–	output: 8 8 (failed)
`a = 5; b = 8;`	5	8	–	start over
`int temp = a;`	5	8	5	the value of a is safe
`a = b;`	8	8	5	
`b = temp;`	8	5	5	
`println`	8	5	5	output: 8 5 (done)

Table 6.26: Trace table of the swap example. The first four rows show the wrong attempt and the later rows show the right method.

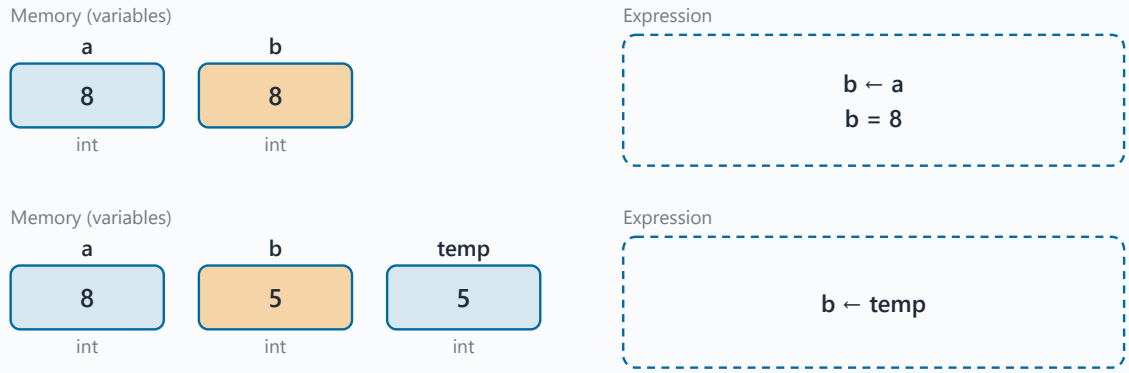


Figure 6.34: Wrong and right swap. Top: after `a = b` and `b = a`, both boxes are 8. Bottom: the temporary variable keeps the 5 and `b = temp` completes the swap.

The problem with the wrong attempt is that an assignment erases the old value. When `a = b` runs, the 5 in a is lost; the following `b = a` only copies 8 back. Just as exchanging the water in two glasses needs a third glass, a swap needs a third box. This box is called a **temporary variable**. The correct method uses three assignments and one temporary variable.

With numbers, a swap can also be done without a temporary variable: `a = a + b; b = a - b; a = a - b;`. For a = 5, b = 8, a first becomes 13, then b = 13 − 8 = 5, and finally a = 13 − 5 = 8. This method is surprisingly robust against overflow:

```
int a = 5, b = 8;
a = a + b;
b = a - b;
a = a - b;
System.out.println(a + " " + b);
int x = 2_000_000_000, y = 1_500_000_000;
x = x + y;
System.out.println("mid: " + x);
y = x - y;
x = x - y;
System.out.println(x + " " + y);
```

```
8 5
mid: -794967296
1500000000 2000000000
```

The sum of two billion and one and a half billion overflows int, and the intermediate value is −794,967,296. The result still comes out right: the 2^{32} added during the overflow is taken back in the following subtractions. Even so, this trick hurts readability and works only with numbers; in everyday programming, the swap with a temporary variable is preferred.

6.6.3 Celsius to Fahrenheit

The formula that converts a temperature in Celsius to Fahrenheit is *F = C · 9 / 5 + 32*. The program reads the temperature from the keyboard and writes the result with one decimal place:

```
Scanner in = new Scanner(System.in);
double c = in.nextDouble();
double f = c * 9 / 5 + 32;
System.out.printf("%.1f C = %.1f F%n", c, f);
```

Input	c	c * 9	… / 5	… + 32 = f	Output
100	100.0	900.0	180.0	212.0	`100.0 C = 212.0 F`
37	37.0	333.0	66.6	98.6	`37.0 C = 98.6 F`

Input	c	c * 9	… / 5	… + 32 = f	Output
-40	-40.0	-360.0	-72.0	-40.0	`-40.0 C = -40.0 F`

Table 6.27: Left-to-right evaluation of the formula in three runs (the values were obtained on JDK 21 by adding intermediate output to the program).

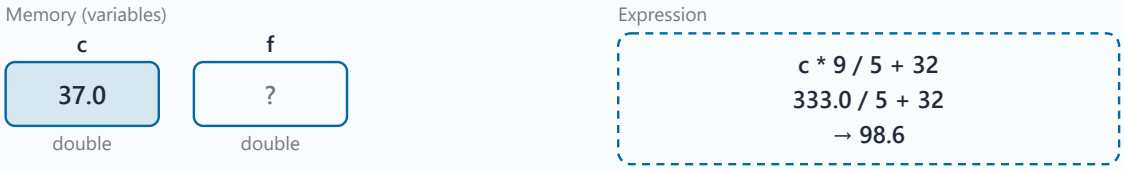


Figure 6.35: The second run: for c = 37.0, *333.0 / 5 + 32 → 98.6* is computed and written into the f box.

Even if the user types 100, `nextDouble()` reads the value as a double: c = 100.0. The formula is evaluated from left to right. `c * 9` is the product of a double and an int and is a double; the division and the addition are floating-point too. The %.1f specifier writes both values with one decimal place. The third run shows an interesting fact: −40 degrees is the same temperature on both scales.

The order in which the formula is written matters. If the same formula were written as `9 / 5 * c + 32`, the first operation would be `9 / 5`. The quotient of two ints is 1, not 1.8. For c = 37 the result comes out as *1 · 37 + 32 = 69* instead of 98.6:

```
double c = 37;
double f1 = 9 / 5 * c + 32;
double f2 = 9.0 / 5 * c + 32;
double f3 = c * 9 / 5 + 32;
System.out.println(f1 + " " + f2 + " " + f3);
System.out.println(9 / 5 + " " + 9.0 / 5);
```

```
69.0 98.60000000000001 98.6
1 1.8
```

When `9.0 / 5` is written, the division becomes floating-point and the result is correct. In this form the value is 98.60000000000001; because the order of operations is different, the floating-point error shows up in another digit. When rounded to one decimal place with printf, both forms give 98.6.

6.6.4 Area and circumference of a circle

```
double r = 2.5;
double area = Math.PI * r * r;
double circ = 2 * Math.PI * r;
System.out.printf("Area: %.2f%n", area);
System.out.printf("Circ: %.2f%n", circ);
```

```
Area: 19.63
Circ: 15.71
```

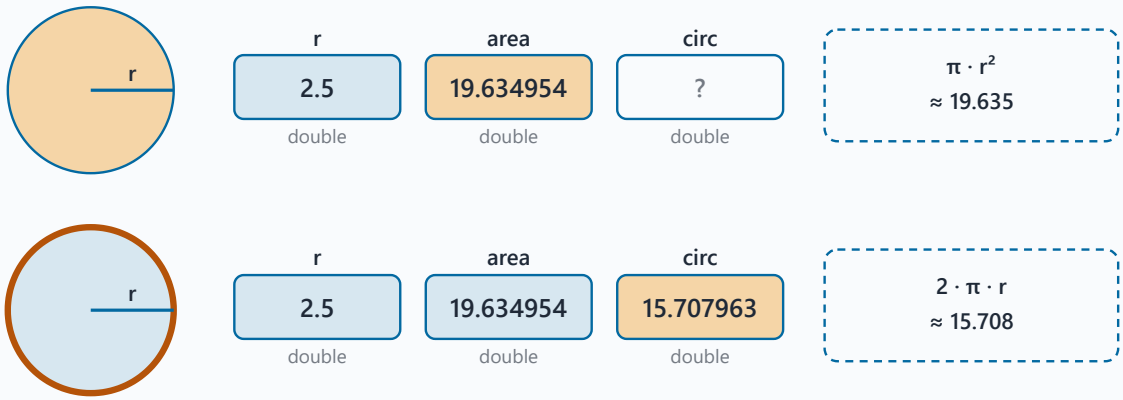


Figure 6.36: For r = 2.5, the area $\pi \cdot r^2 \approx 19.635$ and the circumference $2 \cdot \pi \cdot r \approx 15.708$ are computed and written into the boxes.

The area is $\pi \cdot r \cdot r = 3.14159\ldots \cdot 2.5 \cdot 2.5$, and the value 19.634954084936208 sits in the `area` box. The circumference is $2 \cdot \pi \cdot r$: 15.707963267948966. The value of π is not written by hand as 3.14; `Math.PI` is a ready-made constant with 16 significant digits. The %.2f specifier rounds the values to two decimal places: 19.634… → 19.63 and 15.7079… → 15.71. Rounding affects only printing; the values in the variables do not change, and the full values are used in later computations. `Math.pow(r, 2)` could also be written to square; `r * r` is both shorter and faster.

6.6.5 Converting seconds to hours

Given a duration in seconds, converting it to the H:MM:SS form is a typical problem in which integer division and the remainder operator are used together.

```
int total = 3725;
int hours = total / 3600;
int rem = total % 3600;
int mins = rem / 60;
int secs = rem % 60;
System.out.printf("%d:%02d:%02d%n",
                  hours, mins, secs);
```

```
1:02:05
```

Statement	Computation	Result	Meaning
`hours = total / 3600`	3725 / 3600	1	how many full hours
`rem = total % 3600`	3725 % 3600	125	seconds left over from the hours
`mins = rem / 60`	125 / 60	2	how many full minutes
`secs = rem % 60`	125 % 60	5	seconds left over from the minutes
`printf("%d:%02d:%02d%n", …)`	1, 2, 5	1:02:05	%02d: at least 2 digits, padded with 0 on the left

Table 6.28: Trace table for total = 3725.

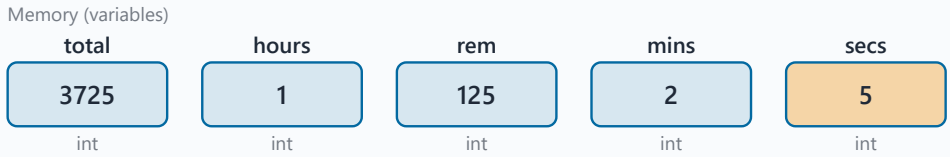


Figure 6.37: The five boxes at the end of the program: total = 3725, hours = 1, rem = 125, mins = 2, secs = 5.

The method proceeds from the large unit to the small one. First the number of full hours is found, then the remaining seconds are split into minutes and seconds. The result is verified with a check: *1 · 3600 + 2 · 60 + 5 = 3725*. The %02d specifier completes single-digit minutes and seconds to two digits by putting a 0 in front; the output is 1:02:05 instead of 1:2:5. The same method works for other durations: 86399 seconds is 23:59:59, and 7384 seconds is 2:03:04. The formulas can also be written on a single line. In the expression `total % 3600 / 60`, % and / have the same precedence and are processed from left to right; first the remainder and then the minutes are found.

Example 6.4

How is an amount given in kuruş (hundredths of a lira) split into lira and kuruş? What is the result for 1575 and 1005 kuruş?

```
int amount = 1575;
System.out.println(amount / 100 + " lira " + amount % 100 + " kuruş");
amount = 1005;
System.out.println(amount / 100 + " lira " + amount % 100 + " kuruş");
System.out.printf("%d,%02d TL%n", amount / 100, amount % 100);
```

```
15 lira 75 kuruş
10 lira 5 kuruş
10,05 TL
```

1 lira is 100 kuruş. *1575 / 100 = 15* full lira, and *1575 % 100 = 75* kuruş are left over. For 1005 kuruş, *1005 / 100 = 10* and *1005 % 100 = 5* are found. When the amount is written in the Turkish way, the kuruş must be padded to two digits; otherwise 10,5 TL would be written instead of 10,05 TL, and the reader would understand 50 kuruş. The %02d in the last line does this padding. Holding the amount in kuruş as an int or long also avoids the floating-point errors of double from the start.

6.7 Common mistakes

Most mistakes made on the topics of this chapter fall into a few patterns. The table in the slides brings five of them together:

Mistake	Example	What happens?
= and == confused	`if (x = 5)`	Compile error: int, not boolean
Integer division	`double o = 7 / 2;`	o = 3.0, not 3.5
Use out of scope	`{ int k = 1; } k++;`	k is undefined outside the block
Wrong semicolon	`if (x > 0);`	Empty statement: the condition controls nothing
String and ==	`s1 == s2`	Compares references, not content → `s1.equals(s2)`

Table 6.29: Common mistakes (the table in the slides).

Confusing = and ==. If a single equal sign is written when equality is meant to be tested in a condition, the expression becomes an assignment. The value of the expression `x = 5` is 5 and its type is int; an if, however, requires a boolean condition. Java catches this mistake at compile time:

```
error: incompatible types: int cannot be converted to boolean
if (x = 5) {
      ^
1 error
```

In languages such as C this line compiles and the condition is always counted as true. The fact that Java does not mix int and boolean turns this mistake into a harmless compile error.

Integer division. In the line `double o = 7 / 2;` the target is a double, but the division is done between two ints: *7 / 2 = 3*. The result widens to double and o = 3.0. It comes from the same root as the `long result = a * b` mistake in the overflow section: the type of an expression depends on the operands, not on where the result will be written.

Use out of scope. A variable is valid throughout the brace block in which it is declared. This region is called the variable's **scope**. When the block closes, the variable ceases to exist:

```
{
    int k = 1;
}
k++;
```

```
error: cannot find symbol
k++;
^
  symbol:   variable k
  location: class Scope
1 error
```

A wrong semicolon. A semicolon placed right after the if condition is a statement in itself for Java: the **empty statement**, which does nothing. The condition controls only this empty statement; the block after it always runs, independently of the condition:

```
int x = -4;
if (x > 0);
{
    System.out.println("positive");
}
```

```
positive
```

Although x = −4, “positive” is printed. The program compiles and gives no warning; that is why this mistake is hard to find.

String and ==. As seen in the section on string concatenation, `==` looks at whether two String variables point to the same object. Content comparison is done with `equals`.

Five mistakes treated in detail in the earlier sections of the chapter can also be added to this list:

- a result that does not fit the type silently overflowing,
- comparing floating-point numbers with `==`,
- thinking that the `(int)` cast rounds,
- thinking that addition will be performed in the expression `"Result: " + 2 + 3`,
- not consuming the line break after `nextInt()`.

6.8 Summary

- A **variable** is a named box in memory set aside for a value of a certain type; it has a name, a type, and a value. A **declaration** specifies the type and name, and **initialization** gives the starting value. A **local variable** must definitely have been assigned a value before it is read; a `final` variable is assigned only once.
- An **assignment** first evaluates the right side and then writes the result into the variable on the left; the old value is erased. A **compound assignment** (`+=`, `-=` ...) updates the variable using its old value and evaluates the right side as a unit.
- An **identifier** starts with a letter, _ or $; it contains no spaces, cannot be a keyword, and is case-sensitive. Variables are named in lowerCamelCase, classes in UpperCamelCase, and constants in UPPER_CASE.
- Java has eight **primitive types**: byte (8), short (16), int (32), long (64), float (32), double (64), char (16), and boolean. The default choices are int and double. long literals need the suffix L and float literals the suffix F.
- Integers are stored in **two's complement** representation. A result that exceeds the range **overflows** and silently wraps around to the other end: `Integer.MAX_VALUE + 1` → −2147483648. To catch overflow, methods such as `Math.addExact` throw an exception. The type of an expression is determined by its operands; `long s = a * b` still multiplies in int.
- **Floating-point numbers** cannot store decimal fractions such as 0.1 exactly: `0.1 + 0.2` is 0.30000000000000004. double values are compared not with `==` but with a **tolerance**. For money, integer hundredths or BigDecimal are used.
- char holds a UTF-16 code unit as a number ('A' = 65); in arithmetic it is promoted to int. boolean takes only true or false and is not interchangeable with numbers.
- **Widening** is automatic and lossless (int → double). **Narrowing** requires an explicit cast: `(int) 9.8` gives 9; the fraction is not rounded but truncated toward zero. In integer narrowing the low bits remain: `(byte) 200` is −56. For rounding, `Math.round` is used.
- The quotient of two integers is **integer division**: `5 / 2` is 2. If one of the operands is a double, the other is **promoted** and `5.0 / 2` is 2.5. `(double) (5 / 2)`, however, does not bring back the lost fraction. byte, short, and char are first promoted to int in operations.
- `%` gives the remainder and takes its sign from the dividend. Dividing an integer by zero throws `ArithmeticException`; floating-point division produces Infinity or NaN.

- The **postfix** `i++` produces the old value and the **prefix** `++i` the new value. Compound assignment implicitly narrows the result to the type of the left side: `b += 5` compiles, `b = b + 5` does not.
- Comparison operators produce booleans; `&&`, `||`, and `!` combine boolean values. **Operator precedence**: parentheses, unary, multiplicative, additive, relational, equality, &&, ||, assignment. Operators at the same level go from left to right. The compiler turns an expression into an **expression tree**.
- **Short-circuit**: in `&&`, if the left side is false, and in `||`, if the left side is true, the right side is not evaluated. The guarding condition (`b != 0`) is written on the left.
- `+` performs **concatenation** if one of the operands is a string, and works from left to right: `"Result: " + 2 + 3` gives “Result: 23” and `"Result: " + (2 + 3)` gives “Result: 5”. String content is compared with `equals`.
- **Scanner** reads from the keyboard; the input waits in a **buffer**. `nextInt()` leaves the line break in the buffer; if a line is to be read next, it is first consumed with a `nextLine()`. Input is validated first for type, then for range.
- **printf** fills the **format specifiers** (%d, %f, %s, %n ...) in the **format string** with the arguments in order; width, precision, and flags (-, 0, ,) align the output. The decimal separator depends on the **locale**; under a Turkish locale, printf writes a comma and nextDouble expects a comma.

6.9 Key terms

The key terms of this chapter are listed below.

variable type declaration initialization assignment local variable constant / literal compound assignment identifier keyword naming convention primitive type reference two's complement sign bit overflow exception floating-point number epsilon (tolerance) string type conversion widening conversion narrowing conversion cast implicit conversion truncation integer division type promotion operator operand expression remainder dividend quotient increment / decrement prefix postfix comparison operator logical operator operator precedence expression tree short-circuit evaluation string concatenation prompt buffer newline input validation formatted output format string format specifier locale decimal separator swap temporary variable scope empty statement

---

6.10 Try it yourself

1. After the lines `int x = 1_000_000; int y = x * 3000; long z = x * 3000L;`, what does

`System.out.println(y + " " + z);` print? Why is y wrong?

Answer

The output is `-1294967296 3000000000`. *1,000,000 · 3,000 = 3,000,000,000* exceeds the int limit, 2,147,483,647. Because both operands are ints, the multiplication is done in int and overflows: *3,000,000,000 − 2^{32} = −1,294,967,296*. In z, 3000L is a long literal; x is promoted to long and the multiplication is done in long.

2. What are the values of the expressions `(byte) 300`, `(int) -7.9`, `Math.round(-7.5)`, and `(int) Math.round(7.5)`?

Answer

44, −7, −7, and 8, respectively. The low 8 bits of 300 are kept: *300 − 256 = 44* (positive because the sign bit is 0). `(int) -7.9` truncates toward zero: −7. Math.round rounds halves in the positive direction: −7.5 → −7, 7.5 → 8; the latter returns a long, so it has been cast to int.

3. After the line `int a = 7, b = 8; double o = (a + b) / 2;`, what is o? What are two forms that give the correct average?

Answer

o = 7.0. `a + b` is 15 and is an int; *15 / 2* becomes 7 by integer division, and then it widens to double and is written as 7.0. The correct forms are `(a + b) / 2.0` and `(double) (a + b) / 2`; both give 7.5. `(double) ((a + b) / 2)`, however, again gives 7.0, because the cast is done after the division.

4. After the lines `int i = 10; int a = i-- - 3; int b = --i * 2;`, what are i, a, and b?

Answer

i = 8, a = 7, b = 16. The postfix `i--` produces the old value (10): *a = 10 − 3 = 7*; then i becomes 9. The prefix `--i` first makes i 8 and then uses the new value: *b = 8 · 2 = 16*.

5. Does `short s = 10; s *= 2.6;` compile? If it does, what is s? What can be said about `s = s * 2.6;`?

Answer

It compiles and s = 26. Compound assignment means `s = (short) (s * 2.6)`: *10 · 2.6 = 26.0* (double) is computed and narrowed to short. `s = s * 2.6;`, however, does not compile; the right side is a double and there is no implicit conversion from double to short ("possible lossy conversion from double to short").

6. What are the values of the expressions `5 + 4 % 3 * 2 - 8 / 3` and `!(3 > 2) || 4 >= 4 && 1 != 1`? In what order is the computation done?

   Answer

   The first is 5. *, /, and % are at the same level and are processed from left to right: *4 % 3 = 1*, *1 · 2 = 2*, *8 / 3 = 2*; then *5 + 2 − 2 = 5*. The second is false. The inside of the parentheses, *3 > 2*, is true, and with ! it becomes false. `&&` comes before `||`: *4 >= 4* is true, *1 != 1* is false, *true && false = false*. The result is *false || false = false*.

7. What do `System.out.println(1 + 2 + "3" + 4 + 5);` and `System.out.println('a' + 2 * 3 + "");` print?

   Answer

   The first prints “3345”: from left to right, first the two ints are added (3), then joined with the string “3” (“33”); after that every operation is concatenation: “334”, “3345”. The second prints “103”: *2 · 3 = 6* is computed first; 'a' is promoted to int and *97 + 6 = 103* is found; finally it is joined with the empty string.

8. The user types `3 4.5` on the first line and `Ali Veli` on the second (Locale.US). The program calls `nextInt()`, `nextDouble()`, `nextLine()`, and `nextLine()` in that order. What do the four calls return?

   Answer

   3, 4.5, "" (empty string), and "Ali Veli". nextInt and nextDouble read the two numbers on the same line; the line break at the end of the first line stays in the buffer. The first nextLine reads up to this line break and returns an empty string. The second nextLine reads the second line. Under a Turkish locale, nextDouble would have thrown an InputMismatchException on the input “4.5”.

9. What does the line `System.out.printf("%-5s|%8.3f|%,d%n", "Paper", 12.34567, 1500000);` print under Locale.US? How many characters does each field take?

   Answer

   `Paper| 12.346|1,500,000`. “Paper” is exactly 5 letters; `%-5s` adds no spaces. `%8.3f` rounds 12.34567 to three decimal places (12.346, 6 characters) and puts 2 spaces in front to complete it to 8 characters. `%,d` adds a thousands separator; under a Turkish locale this field would be 1.500.000 and the second field 12,346.

6.11 Further reading

- R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), 1.2 Built-in Types of Data: explains Java's primitive types, type conversion, and floating-point problems with short programs; the book's site (introcs.cs.princeton.edu) has a summary of the section and exercises.
- Oracle, *The Java Tutorials*, Language Basics: Variables and Operators: gives the primitive types, the way literals are written, the operator precedence table, and compound assignment from the official source, with examples.
- Y. D. Liang, *Introduction to Java Programming and Data Structures* (13th ed., Pearson, 2024), Chapter 2 Elementary Programming and Chapter 4: contains many solved examples on input with Scanner, numeric types, printf, and the Math class.
- Oracle, *Java Language Specification*, Java SE 21: Chapter 5 (Conversions and Contexts) and Chapter 15 (Expressions): the precise definitions of the conversion, promotion, and evaluation-order rules described in this chapter; the last word on any advanced detail you may wonder about is here.

6.12 References

1. P. Deitel, H. Deitel, *Java: How to Program, Early Objects*, 11th ed., Pearson, 2018.
2. Y. D. Liang, *Introduction to Java Programming and Data Structures*, 13th ed., Pearson, 2024.
3. Oracle, *The Java Tutorials*: Variables, Operators.
4. Oracle, *Java Language Specification, Java SE 21 Edition*: Chapter 4, Types, Values, and Variables; Chapter 15, Expressions.
5. Oracle, *Java SE 21 API*: java.util.Scanner, java.lang.Math, java.util.Formatter (format specifiers).
6. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 1.2 Built-in Types of Data.

Conditionals

So far, the Java programs we have written execute the same lines in the same order every time they run. Real programs behave differently depending on the situation. A score below 60 prints “Fail”; otherwise the program prints “Pass”. A division is skipped when the divisor is zero. A warning appears when the user enters month 13. This chapter describes the constructs that let a program look at a condition and choose one of several paths. We begin with sequential execution and the idea of a condition. Next come `if`, `if-else`, the `else-if` chain, and nested decisions. Logical operators, short-circuit evaluation, the conditional operator, and string comparison let us write decisions more compactly and more safely. The `switch` construct selects among the discrete values of a single expression. The examples in the last section combine all of these constructs in realistic problems. Every program was compiled and run with JDK 21; the outputs shown are the programs' actual outputs.

LEARNING OUTCOMES

Flow
Distinguish sequence, selection, and repetition structures; show in which order and under which conditions the lines of a program run by using a trace table.

Conditions
Write conditions that produce boolean values with comparison and logical operators; use short-circuit evaluation, De Morgan's laws, and operator precedence correctly.

Decisions
Build `if`, `if-else`, `else-if` chains and nested decisions in the right order; explain the difference between independent ifs and a chain, the dangling else problem, and the role of braces.

switch
Use the `switch` expression and statement for the discrete values of a single expression; explain `default`, `yield`, `break`, and fall-through; choose between if-else and switch.

7.1 Control flow

A program is a list of statements given to the computer. A **statement** is the smallest complete unit of a program that does one job: declaring a variable, assigning a value, printing a line. In Java every simple statement ends with a semicolon. The order in which the computer executes these statements is called **control flow**. The language constructs that determine the flow are collectively called **control structures**.

7.1.1 The three basic flows

However large an algorithm is, it can be written with only three forms of flow. Figure 7.1 shows the three side by side.

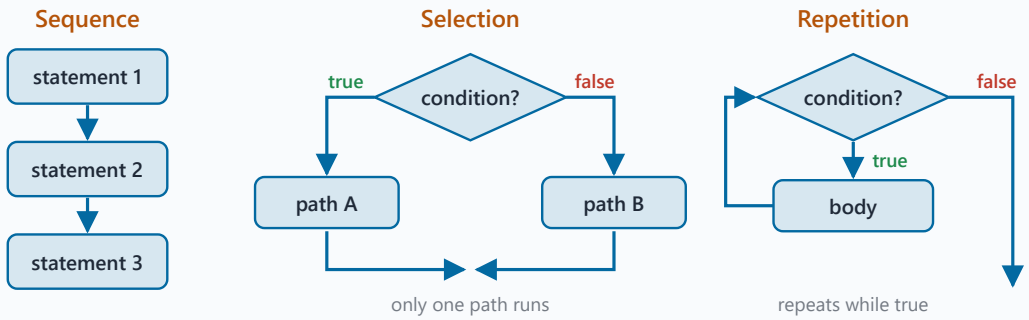


Figure 7.1: The three basic flows. Left: in a sequence, statements run once each, in the order written. Middle: in a selection, only one of the two paths runs, depending on the value of the condition. Right: in a repetition, the body repeats as long as the condition stays true.

Definition 7.1

A **sequence structure** is a flow in which statements run exactly once each, in the order written. A **selection structure** is a flow in which one of several paths is chosen according to the value of a condition; it is also called a conditional structure. A **repetition structure** is a flow in which a block of code runs over and over for as long as a condition holds.

Each of these three structures has a single entry and a single exit, so they can be nested inside one another. A branch of a decision can contain a sequence of statements, and the body of a loop can contain a decision. This chapter treats selection in detail. A sequence is the building block of a decision, so we examine it first.

7.1.2 Sequence

The following four lines compute the area of a triangle. The base is 8 units and the height is 5 units.

```
double base = 8;
double height = 5;
double area = base * height / 2;
System.out.println(area);
```

The lines run from top to bottom. The first line creates the variable `base` and stores 8 in it. Because the variable's type is `double`, the integer 8 is converted automatically to 8.0. A conversion from a smaller type to a larger type is called a **widening conversion**. The second line does the same for `height`. The third line uses the values of the previous two variables and computes the area. The fourth line prints the result.

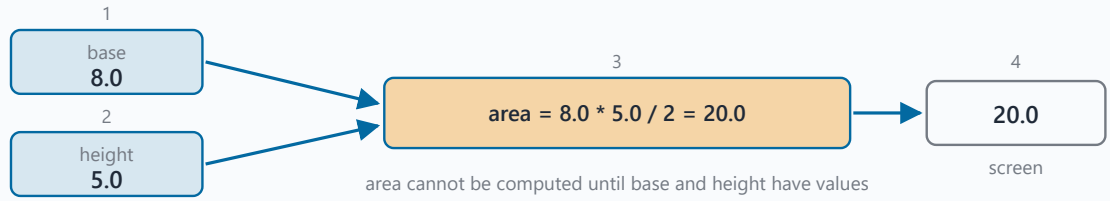


Figure 7.2: A sequence. Lines 1 and 2 give values to base and height. Line 3 uses these two values to compute the area as 20.0. Line 4 prints the result.

Figure 7.2 uses arrows to show the dependencies between lines. The third line needs the values produced by the first and second lines. Every statement can use the state left behind by the statements before it. That is the essence of a sequence.

7.1.3 Tracing a sequence step by step

The most reliable way to execute a program by hand is a trace table. In a **trace table**, each row shows one step and each column shows one variable. A cell holds the variable's value when that step has finished. A variable that does not exist yet is shown as "—".

Step	Line executed	base	height	area	Screen
1	`double base = 8;`	8.0	—	—	
2	`double height = 5;`	8.0	5.0	—	
3a	`base * height`	8.0	5.0	— (intermediate result 40.0)	
3b	`40.0 / 2` → `area`	8.0	5.0	20.0	
4	`System.out.println(area);`	8.0	5.0	20.0	20.0

Table 7.1: Trace table of the area computation. Each row shows the state after the corresponding statement has finished.

The third line is examined in two steps. `*` and `/` have the same precedence and are evaluated from left to right. First *8.0 × 5.0 = 40.0* is found, then *40.0 / 2 = 20.0*. Only when both steps are finished is a value written into the `area` box. In the middle of an expression, the intermediate result exists only inside the processor; it is not stored in any variable. When the program runs, it produces this output:

```
20.0
```

In a sequence, the position of a line determines the result. If the third line is moved to the very top, `base` and `height` do not exist yet at that point. Java catches this error at compile time, before the program runs:

```
public class OrderError {
    public static void main(String[] args) {
        double area = base * height / 2;
        double base = 8;
        double height = 5;
        System.out.println(area);
    }
}
```

```
OrderError.java:3: error: cannot find symbol
        double area = base * height / 2;
                      ^
  symbol:   variable base
  location: class OrderError
OrderError.java:3: error: cannot find symbol
        double area = base * height / 2;
                             ^
  symbol:   variable height
  location: class OrderError
2 errors
```

The message means “symbol not found”. On line 3 the compiler looks for a variable named `base` and cannot find it. The same message is given for `height`. Java does not allow a variable to be used before the line on which it is declared.

Common mistake

Reading the formula “area = base × height / 2” as a definition, as in mathematics. In Java, `=` is not a declaration of equality but an assignment: the expression on the right is evaluated at that moment and the result is written into the box on the left. Even if `base` changes after the assignment, `area` is not updated automatically. The assignment has to run again to produce a

new area.

7.2 Conditions and if

At the heart of a selection structure is a question: “Is the score 60 or more?”, “Is the number negative?”, “Is the divisor zero?” Which path the program takes depends on the answer. In Java, this question is written as a condition.

7.2.1 Conditions: expressions that produce a boolean

Definition 7.2

A **condition** is an expression whose value is `true` or `false`. The Java type that holds these two values is called **boolean**. Only an expression that produces a boolean value may appear in the parentheses of an `if` statement.

Most conditions are built by comparing two values. Java's six **comparison operators** are listed in Table 7.2. Each takes two numbers and produces a boolean value.

Operator	Meaning	Example	Value	Opposite
`==`	equal	`x == 5`	false	`!=`
`!=`	not equal	`x != 5`	true	`==`
`<`	less than	`x < 7`	false	`>=`
`<=`	less than or equal	`x <= 7`	true	`>`
`>`	greater than	`x > 5`	true	`<=`
`>=`	greater than or equal	`x >= 9`	false	`<`

Table 7.2: Comparison operators. In the Example column, x = 7.

The last column of the table gives the opposite of each operator. The opposite of `>` is not `<` but `<=`, because “not greater than 7” means “7 or less”, and 7 itself belongs to that set. This detail is the source of a common mistake when negating conditions.

Equality is tested with two equal signs (`==`). A single equal sign (`=`) is an assignment. Confusing the two is one of the most common mistakes beginners make. Fortunately, Java catches this confusion at compile time:

```
public class AssignInCondition {
    public static void main(String[] args) {
        int x = 7;
        if (x = 5) {
            System.out.println("five");
        }
    }
}
```

```
AssignInCondition.java:4: error: incompatible types: int cannot be converted to boolean
        if (x = 5) {
              ^
1 error
```

`x = 5` is an assignment. The value of an assignment is the number assigned, here 5, and its type is `int`. An `if`, however, expects a boolean. That is why the compiler reports "int cannot be converted to boolean". In some languages, such as C, any nonzero number counts as "true" and this line would compile silently. In Java, numbers and booleans are separate types; this strictness eliminates a whole class of errors.

The mathematical notation *10 < x < 20* does not work in Java either. The expression is read from left to right: first `10 < num` is evaluated and yields a boolean value. Then the program tries to compare that boolean value with 20. "Less than" is meaningless for a boolean:

```
public class ChainedComparison {
    public static void main(String[] args) {
        int num = 15;
        if (10 < num < 20) {
            System.out.println("between");
        }
    }
}
```

```
ChainedComparison.java:4: error: bad operand types for binary operator '<'
        if (10 < num < 20) {
                     ^
  first type:  boolean
  second type: int
1 error
```

The message says that the `<` operator has a boolean on one side and an int on the other. The correct form combines the two comparisons with a logical AND: `num > 10 && num < 20`. Logical operators are treated in detail in Table 7.9.

7.2.2 The one-way if

The simplest decision does a job if the condition is true and does nothing if it is false. For example, to turn a negative number into a positive one, an operation is needed only when the number is negative:

```
if (num < 0) {
    num = -num;
}
```

When execution reaches the `if` line, the expression in the parentheses is evaluated first. If the result is `true`, the statements between the braces run. If the result is `false`, those statements are skipped and execution continues at the line after the closing brace. In both cases the program reaches the next line; the work in between is either done or not. This pattern is traced in detail in the absolute value example.

7.2.3 The if-else statement

Many decisions have a separate job for each of two cases. If the score is 60 or more, the program must print “Pass”; otherwise it must print “Fail”. For this we add an `else` branch to the `if`.

Definition 7.3

An **if-else statement** consists of two branches. If the condition is `true`, the body of the `if` runs; if it is `false`, the body of the `else` runs. On every execution exactly one of the branches runs; never both, and never neither. A **branch** is each of the paths that can be followed after a decision.

```
if (score >= 60) {
    System.out.println("Pass");
} else {
    System.out.println("Fail");
}
```

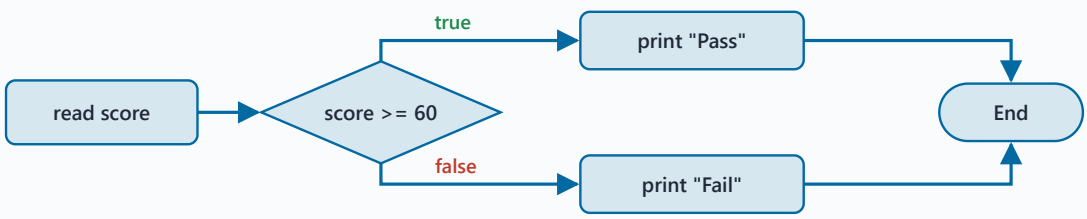


Figure 7.3: Flowchart of the if-else statement. After the score is read, the condition `score >= 60` is asked. The true branch prints “Pass” and the false branch prints “Fail”. The two branches join again at End.

In Figure 7.3, two arrows leave the decision box. One is labeled true and the other false. For any given score, only one of the two can be followed. At the ends of the arrows the two paths merge again; the program continues from the same point after the decision.

The boundary value in the condition needs care. With `score >= 60`, 60 falls on the "Pass" side. If `score > 60` had been written, a student with exactly 60 would fail. The most common error in a decision is putting the boundary on the wrong side. For this reason a decision is always tested with the boundary value and with its two neighbors (such as 59 and 61). Every possible value goes to exactly one branch: scores from 0 to 59 go to the else branch, and 60 and above go to the if branch.

7.2.4 Tracing if-else step by step

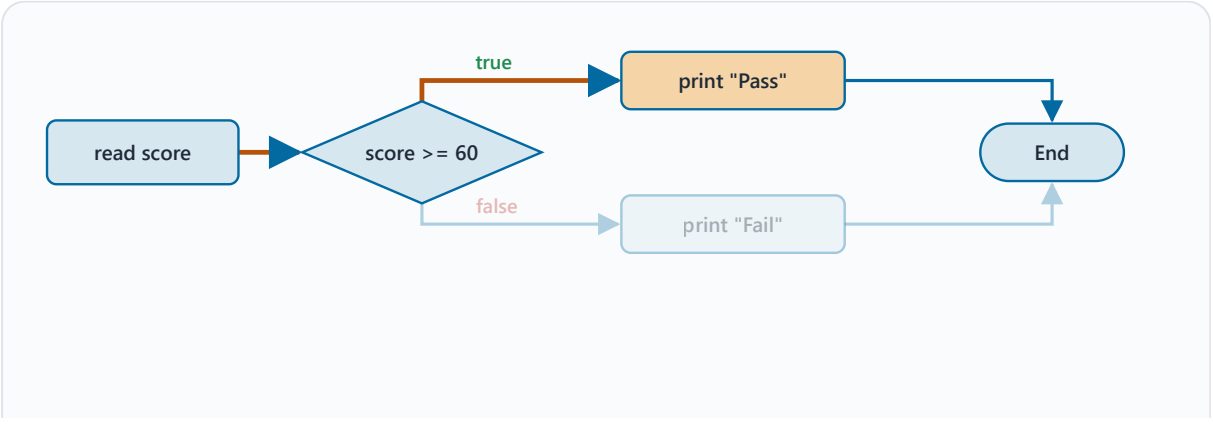
The following program reads the score from the keyboard. The `Scanner` class is in the `java.util` package; to use it, the file begins with the line `import java.util.Scanner;`. From here on, to keep code fragments short, only the body of `main` is shown. The complete program is shown separately.

```
Scanner in = new Scanner(System.in);
int score = in.nextInt();
if (score >= 60) {
    System.out.println("Pass");
} else {
    System.out.println("Fail");
}
```

Let the program run three times, with the user entering 75, 60, and 45 in turn. Table 7.3 gives the steps of each run.

Input	Line 2: score	Line 3: condition	Branch executed	Branch skipped	Output
75	75	*75 >= 60* → true	line 4	line 6	Pass
60	60	*60 >= 60* → true	line 4	line 6	Pass
45	45	*45 >= 60* → false	line 6	line 4	Fail

Table 7.3: Three runs of the Pass/Fail program. The line numbers are the lines of the code above.



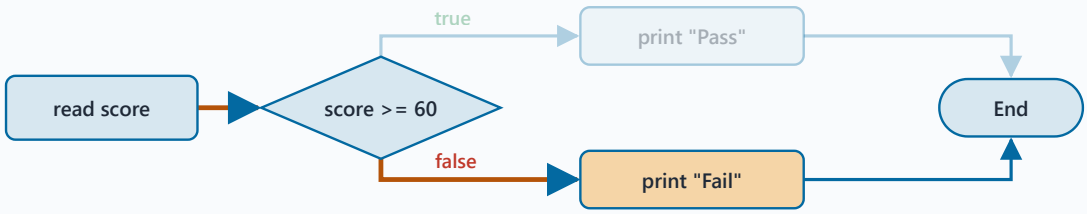


Figure 7.4: The path followed in two runs. (a) score = 75: the condition is true, the "Pass" box runs, and the "Fail" box stays faded. (b) score = 45: the condition is false and only the "Fail" box runs.

The second run tests the boundary value. *60 >= 60* is true because `>=` includes equality. Figure 7.4 uses color to show the path followed in two runs. In every run, one of the branches does not run at all. The complete program and the outputs obtained with the inputs 75, 60, and 45 are as follows:

```
import java.util.Scanner;

public class PassFail {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int score = in.nextInt();
        if (score >= 60) {
            System.out.println("Pass");
        } else {
            System.out.println("Fail");
        }
    }
}
```

```
Pass
```

```
Pass
```

```
Fail
```

7.2.5 Blocks, braces, and the empty statement

A group of statements between braces is called a **block**. Java's rule is this: the body of an `if` and the body of an `else` is a single statement. A block counts as one statement, however many statements it contains. When the body is a single simple statement, the braces may be omitted:

```
if (score >= 60)
    System.out.println("Pass");
else
    System.out.println("Fail");
```

This form is short, but it has two traps. The first appears when a second line is added to the body. The indentation suggests that the added line also belongs to the `if`, but the compiler does not look at indentation. Without braces, the `if` controls only the single statement that immediately follows it; the second line runs in every case.

```
public class NoBraces {
    public static void main(String[] args) {
        int x = 2;
        if (x > 5)
            System.out.println("A");
            System.out.println("B");
        System.out.println("C");
    }
}
```

```
B
C
```

For x = 2, *2 > 5* is false. Only `println("A")` is skipped. Although the “B” line is indented, it is outside the `if` and it runs. If the programmer meant to control both A and B, the two lines must be enclosed in braces.

The second trap is a semicolon accidentally placed at the end of the `if` line. A lone semicolon is also a statement in Java: the **empty statement**, which does nothing. When you write `if (x > 0);`, the body of the `if` becomes this empty statement. The line below is no longer controlled by the condition.

```
public class EmptyStatement {
    public static void main(String[] args) {
        int x = -3;
        if (x > 0);
            System.out.println("Positive");
        System.out.println("Done");
    }
}
```

```
Positive
Done
```

Although x = −3, “Positive” is printed. *−3 > 0* is false and the empty statement is skipped, but there is nothing else to skip. The compiler does not count this line as an error because nothing breaks the rules. The mistake shows up only in the program's output.

Common mistake

Adding a second line to the body of a one-line `if` later and forgetting the braces. Because the indentation looks right, the error goes unnoticed. For this reason many coding guidelines require braces even when the body is a single line. Braces also make the `if (...);` mistake visible: a line such as `if (x > 0); {` stands out when read.

7.3 Multiway decisions

Decisions with two options are built with if-else. What happens when there are more than two options? Converting a score to a letter grade needs four options: 90 and above is A, 80–89 is B, 70–79 is C, and below 70 is F. A decision of this kind is called a **multiway decision**.

7.3.1 The else-if chain

Definition 7.4

An **else-if chain** is a multiway decision built by putting a new `if` in the body of each `else`. The conditions are tried in order from top to bottom. The branch of the first condition that is `true` runs; the rest of the chain is skipped without being checked. If no condition holds, the final `else`, if there is one, runs.

```
if (score >= 90) {
    grade = 'A';
} else if (score >= 80) {
    grade = 'B';
} else if (score >= 70) {
    grade = 'C';
} else {
    grade = 'F';
}
```

Seeing that the chain actually consists of nested if-else statements makes the rule easier to understand. The body of the first `else` is the second `if`; the body of its `else` is the third `if`. Because the code would shift a little further right at every level, Java programmers write `else if` on the same line and keep all the branches aligned. As far as the language is concerned, there is no separate keyword `else if`; it is only a writing habit.

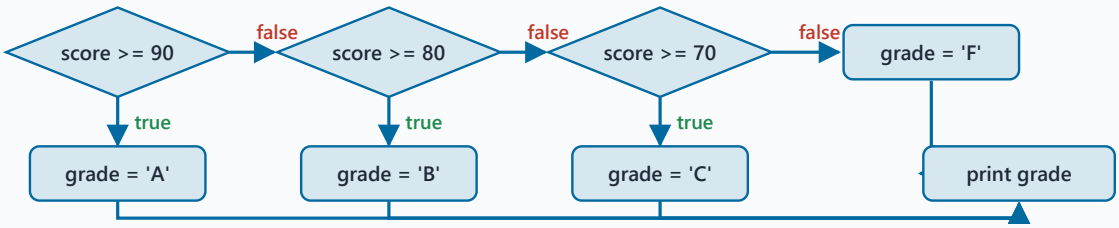


Figure 7.5: Flowchart of the else-if chain. The thresholds 90, 80, and 70 are tested in order from left to right. When a condition is true, that branch's assignment is made and execution goes directly to the "print grade" box. If all three conditions are false, grade = 'F'.

Figure 7.5 shows the shape of the chain. The false arrow of each decision box leads to the next decision. The true arrows go down to the assignment boxes. All paths finally merge at the "print grade" box. In one run, at most one assignment box runs.

7.3.2 Tracing the chain step by step

Here is the short form of the letter-grade program. Because the branches are single statements, no braces are used; the conditions and assignments are aligned to keep the code readable.

```
int score = in.nextInt();
char grade;
if (score >= 90)      grade = 'A';
else if (score >= 80) grade = 'B';
else if (score >= 70) grade = 'C';
else                  grade = 'F';
System.out.println(grade);
```

The `char grade;` on the second line declares the variable but gives it no value. One of the branches of the chain assigns the value. Let the program run for 75, 95, and 50.

score	`score >= 90`	`score >= 80`	`score >= 70`	Branch executed	Condition tested	Output
75	false	false	true	`grade = 'C'`	3	C
95	true	skipped	skipped	`grade = 'A'`	1	A
50	false	false	false	`grade = 'F'` (else)	3	F

Table 7.4: Three runs of the else-if chain. "skipped" means the condition is never evaluated.

For 75, the first two conditions are false: *75 >= 90* and *75 >= 80*. The third condition, *75 >= 70*, is true, so `grade = 'C'` is executed. Even if no further conditions remained in the chain, the rule would be the same: after the chosen branch, execution jumps directly to the end of the chain. For 95, the first condition is true; the other two conditions are never evaluated. For 50, all three conditions are false and the `else` runs.

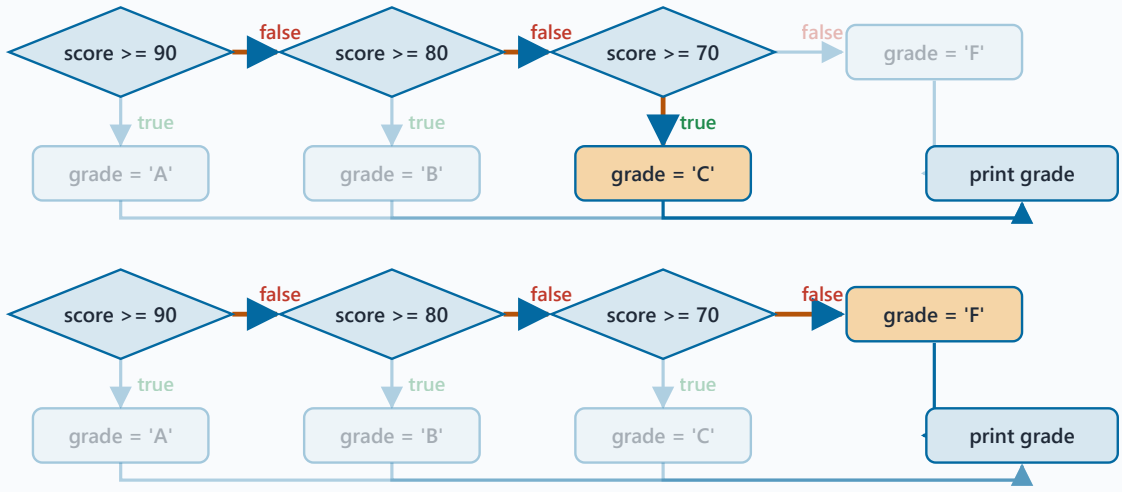


Figure 7.6: The path followed in the chain. (a) score = 75: after two false arrows, the third condition is true and the grade = 'C' box runs. (b) score = 50: all three conditions are false; the path reaches the rightmost grade = 'F' box.

The boundaries of the chain should be tested too. The program was also run for 90, 89, 80, 70, and 69. It printed A, B, B, C, and F, respectively. A difference of a single unit between 90 and 89, or between 70 and 69, changes the grade; each threshold includes its own value.

7.3.3 The order of thresholds

Each condition in the chain sets a lower bound on its own: “at least 80”, “at least 70”. No upper bound is written. The condition of the B branch does not have to be `score >= 80 && score < 90`, because execution reaches B's condition only if A's condition is false; at that point it is already known that the score is less than 90. In a chain, every condition automatically carries the knowledge that the previous conditions were false.

This convenience carries a responsibility: the thresholds must be written from largest to smallest. The following program shows what happens when the order is reversed.

```
import java.util.Scanner;

public class ReversedThresholds {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int score = in.nextInt();
        if (score >= 60) System.out.println("D");
        else if (score >= 70) System.out.println("C");
        else if (score >= 80) System.out.println("B");
        else if (score >= 90) System.out.println("A");
        else System.out.println("F");
    }
}
```

```
D
```

For 85, the first condition *85 >= 60* is true and D is printed. The rest of the chain is skipped. The program also prints D for 95. Every score of 60 and above gets a D. The C, B, and A branches cannot be reached with any input. Such code does not produce a compile error; the mistake shows up only when it is tested with an input whose correct output is known.

Common mistake

Arranging the thresholds from smallest to largest in a chain built on lower bounds. The general rule is: ask the narrower condition first. Every score that satisfies “at least 90” also satisfies “at least 70”. If the broader condition is asked first, the narrower one never gets its turn.

7.3.4 Independent ifs versus an else-if chain

If the same three conditions are written one under another without the `else` keywords, a completely different program results. This form is called **independent ifs**: none of them is in the `else` branch of the previous one, and each is checked independently of the others.

```
int score = 95;
if (score >= 90) System.out.println("A");
if (score >= 80) System.out.println("B");
if (score >= 70) System.out.println("C");
```

```
A
B
C
```

For score = 95, all three conditions are true and three lines are printed. The first `if` printing A does not affect the second `if`; the second `if` is still tested. One expects a student to receive a single letter, but three letters appear. When the same conditions are linked with `else if`, only the first true branch runs:

```
int score = 95;
if (score >= 90) System.out.println("A");
else if (score >= 80) System.out.println("B");
else if (score >= 70) System.out.println("C");
```

```
A
```

Conditions	Independent ifs	else-if chain
`score >= 90`	tested: true → A	tested: true → A
`score >= 80`	tested: true → B	not tested
`score >= 70`	tested: true → C	not tested
Output	3 lines: A, B, C	1 line: A

Table 7.5: Comparison of the two forms for score = 95.

When the branches assign to a variable instead of printing, the error is more insidious. In Figure 7.7, every branch assigns a value to `grade`. In the independent form, a later assignment overwrites an earlier one. Because only a single letter is visible on the screen, the output “looks right”, but the grade is wrong.

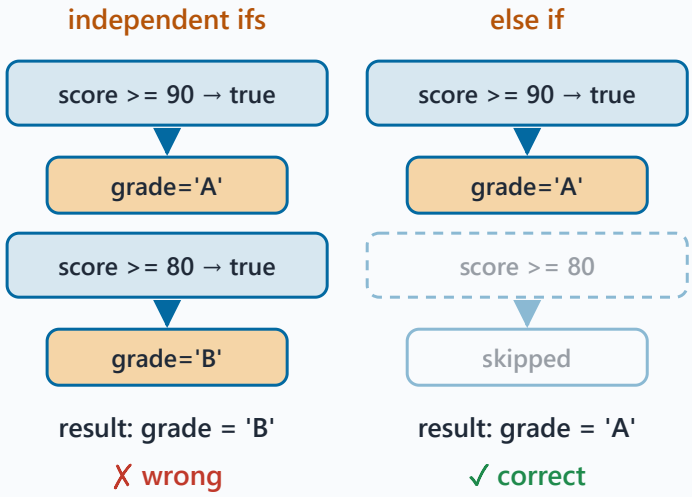


Figure 7.7: Two forms for score = 95. Left, independent ifs: both conditions are true, the second assignment overwrites the first, and grade = 'B' remains. Right, else-if: when the first condition is true, the second is skipped and grade = 'A' remains.

```
public class IndependentGrade {
    public static void main(String[] args) {
        int score = 95;
        char grade = 'F';
        if (score >= 90) grade = 'A';
        if (score >= 80) grade = 'B';
        System.out.println(grade);
    }
}
```

```
B
```

Independent ifs are not always wrong. When each of several independent questions is to be asked separately, it is the right structure. For example, on an application form, separate warnings are written for “Is the name empty?”, “Is the e-mail invalid?”, and “Is the age under 18?” One being true should not prevent the others from being asked. The rule of choice is this: use `else if` when only one of the options can apply, and independent `if`s when each can apply on its own.

7.3.5 The final else and a variable that was never assigned

The `else` at the end of the chain catches the case in which no condition holds. This branch is not mandatory. But if it is deleted from the letter-grade program, the Java program does not compile:

```
public class UnassignedGrade {
    public static void main(String[] args) {
        int score = 75;
        char grade;
        if (score >= 90)      grade = 'A';
        else if (score >= 80) grade = 'B';
        else if (score >= 70) grade = 'C';
        System.out.println(grade);
    }
}
```

```
UnassignedGrade.java:8: error: variable grade might not have been initialized
        System.out.println(grade);
                           ^
1 error
```

The message means “the variable grade might not have been given a value”. The compiler examines all paths. When all three conditions are false, no value is assigned to `grade`. If execution reached the `println(grade)` line along that path, there would be no value to print. Java does not allow an unassigned local variable to be read. This rule prevents programs in which “it is unclear what the variable holds” already at compile time.

The problem can be solved in two ways: either the final `else` is put back, or the variable is given an initial value when it is declared (`char grade = 'F';`). The first way is clearer: the code shows one by one which letter each possible score goes to.

7.4 Nested decisions

Some questions make sense only after the answer to another question is known. For permission to drive a vehicle, age is checked first. If the person is under 18, there is no point asking whether they have a license. If the age is sufficient, the second question is asked: do they have a license?

7.4.1 A decision inside a branch

> **Definition 7.5**
>
> A **nested decision** is a selection structure placed inside one of the branches of another selection structure. The inner condition is checked only when the outer condition selects that branch.

```
int age = in.nextInt();
boolean license = in.nextBoolean();
if (age >= 18) {
  if (license)
    System.out.println("Can drive");
  else
    System.out.println("Get a license first");
} else {
  System.out.println("Too young");
}
```

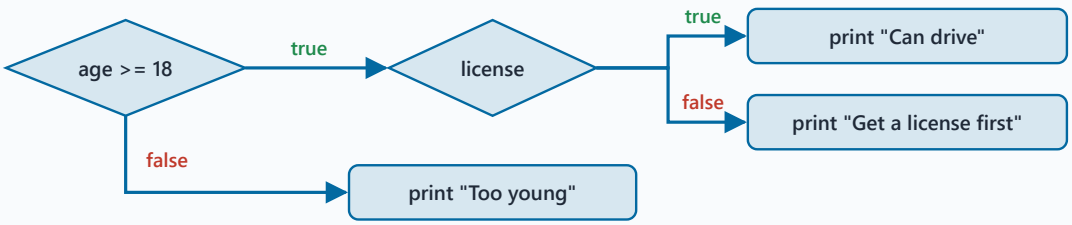


Figure 7.8: A nested decision. First `age >= 18` is asked. The true branch contains a second decision, `license`, which prints two different messages. The false branch prints “Too young” directly.

Figure 7.8 shows the structure. There is a second decision box in the true branch of the outer decision. This box is reached only when the age condition is satisfied. The program has three possible outputs, each at the end of a separate path.

In the code, `nextBoolean()` reads the word `true` or `false` from the keyboard and converts it to a boolean value. The form `if (license)` is also worth noting. `license` is already a boolean variable; there is no need to write `if (license == true)`. The variable itself is the condition.

7.4.2 Tracing a nested decision step by step

Let the program run with three input pairs: (20, true), (20, false), and (15, true).

age	license	Outer condition `age >= 18`	Inner condition `license`	Output
20	true	*20 >= 18* → true	true	Can drive
20	false	*20 >= 18* → true	false	Get a license first
15	true	*15 >= 18* → false	not checked	Too young

Table 7.6: Three runs of the nested decision.

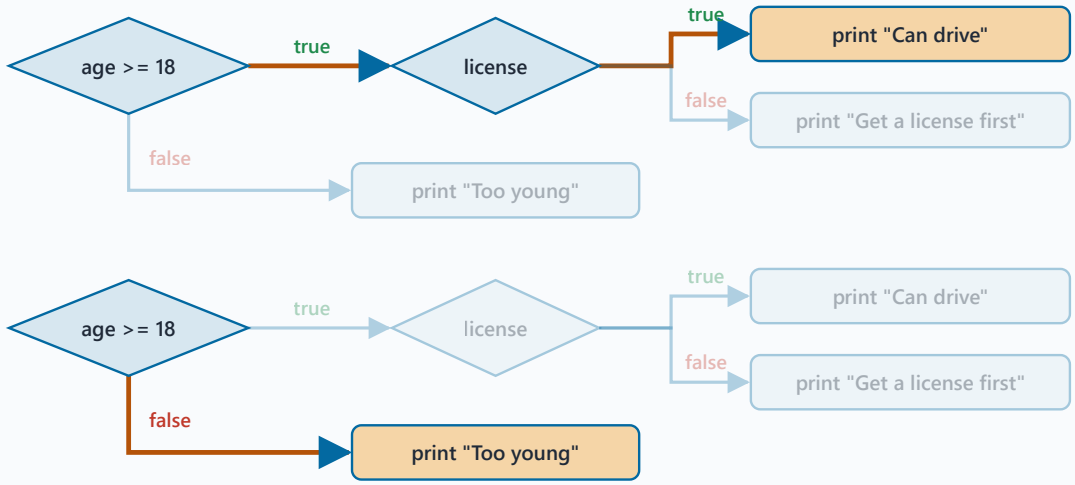


Figure 7.9: The path followed in a nested decision. (a) age = 20, license = true: both conditions are true and "Can drive" is printed. (b) age = 15: the outer condition is false; the inner decision box stays faded because it is never checked.

The third run shows what nesting means. The 15-year-old's license was entered as `true`; in real life this is impossible, but it does not matter to the program. When the outer condition is false, the inner `if` line is never reached. The actual outputs of the complete program:

```
import java.util.Scanner;

public class License {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int age = in.nextInt();
        boolean license = in.nextBoolean();
        if (age >= 18) {
          if (license)
            System.out.println("Can drive");
          else
            System.out.println("Get a license first");
        } else {
          System.out.println("Too young");
        }
    }
}
```

```
Can drive
```

```
Get a license first
```

```
Too young
```

7.4.3 Using && instead of nesting

A nested structure shifts the code a little further right at every level. At a depth of three or four levels, it is hard to follow which `else` belongs to which `if`. In many cases, nested decisions can be turned into a flat chain by combining them with the logical AND (`&&`):

```
import java.util.Scanner;

public class LicenseAnd {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int age = in.nextInt();
        boolean license = in.nextBoolean();
        if (age >= 18 && license)
            System.out.println("Can drive");
        else if (age >= 18)
            System.out.println("Get a license first");
        else
            System.out.println("Too young");
    }
}
```

The first condition catches the case “18 or older and licensed”. The second condition is reached only if the first is false. At that point, if `age >= 18` is true, the person has no license. Every remaining case is insufficient age. For four input pairs, the program gave the same output as the nested version:

Input	License (nested)	LicenseAnd (&&)
20 true	Can drive	Can drive
20 false	Get a license first	Get a license first
15 true	Too young	Too young
15 false	Too young	Too young

Table 7.7: Outputs of the nested version and the && version for four inputs.

Which form to choose depends on readability. If the second question really depends on the first, that is, if the second question is meaningless when the first condition is false, the nested structure shows this dependency clearly. If the conditions carry equal weight, combining them on a single line with `&&` is shorter and flatter.

7.4.4 The dangling else

When braces are not used in nested ifs, an ambiguity arises. In the following code there is a single `else` but two `if` s. Which one does the `else` belong to?

```
int a = -1, b = 5, x = 0;
if (a > 0)
    if (b > 0)
        x = 1;
    else
        x = 2;
System.out.println(x);
```

Definition 7.6

The **dangling else** is the situation in nested ifs without braces in which the notation does not make clear which `if` an `else` is attached to. Java's rule is definite: an `else` is attached to the *nearest* preceding `if` that does not yet have an `else` . Indentation does not change this rule.

In the code, the `else` is written aligned with the inner `if` ; here the indentation agrees with the rule. Even so, a programmer might take the `else` to belong to the outer `if` . Let the initial values be a = −1, b = 5, x = 0. The steps are as follows:

1. `a > 0` is evaluated: *−1 > 0* is false.
2. The body of the outer `if` is skipped. This body is a single statement consisting of the inner `if` and its `else` . Therefore the `else` is skipped as well.
3. `x` never changes; 0 is printed.

```
0
```

A programmer who believes the `else` belongs to the outer `if` expects 2. In reality the `else` is attached to the inner `if` , and since the inner `if` is never reached, it does not run either. The rule becomes even clearer when the same code is tried with different inputs:

a	b	x (initially)	`a > 0`	`b > 0`	Assignment executed	Output
−1	5	0	false	not checked	none	0
−4	6	9	false	not checked	none	9
3	5	0	true	true	`x = 1`	1

a	b	x (initially)	`a > 0`	`b > 0`	Assignment executed	Output
3	−2	0	true	false	`x = 2` (inner else)	2

Table 7.8: Running the dangling else code with four inputs. The program has been modified to read the initial values of a, b, and x from the keyboard.

The assignment `x = 2` runs only when a is positive and b is not. This is proof that the `else` belongs to the inner `if`. The result does not change even if the `else` line is moved all the way to the left, aligned with the outer `if`. The compiler does not read whitespace.

If the programmer's intention is to attach the `else` to the outer `if`, the inner `if` is closed with braces. The closing brace ends the inner `if`; the `else` that follows can no longer attach to it.

```
int a = -1, b = 5, x = 0;
if (a > 0) {
    if (b > 0) x = 1;
} else {
    x = 2;
}
System.out.println(x);
```

```
2
```

With the same initial values, *−1 > 0* is false. This time the `else` belongs to the outer `if` and it runs: `x = 2`, and 2 is printed. The only difference between the two programs is a pair of braces; the outputs are 0 and 2.

Common mistake

Working out what the code does by looking at the indentation. Indentation is meant only for the reader; to the Java compiler, spaces and line breaks are meaningless. Writing every body with braces in nested ifs removes the ambiguity entirely. The “format code” command of development environments also rearranges the indentation to match the structure the compiler sees; a dangling `else` moves to its real place after formatting.

7.5 Decisions with logical operators

Real decisions often depend on more than one condition. An application is accepted if the person is 18 or older *and* has a document. A day is part of the weekend if it is Saturday *or* Sunday. The Java counterparts of the connectives “and”, “or”, and “not” are the logical operators.

7.5.1 &&, || and !

Definition 7.7

Logical operators combine boolean values and produce a new boolean value. `a && b` (AND) is `true` only when both sides are `true`. `a || b` (OR) is `true` when at least one side is `true`. `!a` (NOT) reverses the value.

a	b	`a && b`	`a || b`	`!a`
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Table 7.9: Truth table of the logical operators. The four rows are all combinations of two boolean values.

The meaning of `||` differs from the "either … or …" of everyday language. In everyday speech, "either tea or coffee" often means exactly one of the two. `||`, however, also gives `true` when both sides are true. The application example is written with `&&` as follows:

```
boolean eligible = age >= 18 && hasDoc;
if (eligible) {
    System.out.println("Application accepted");
}
```



Figure 7.10: The values of `age >= 18` and `hasDoc` combine through `&&` to form `eligible`. In the table on the right, of the four combinations only the cell where both are true gives true.

The result of the condition is first assigned to a boolean variable named `eligible`. This is not required; the condition could also have been written directly in the parentheses of the `if`. But giving the condition a name makes the code readable: the line `if (eligible)` explains itself.

7.5.2 Short-circuit evaluation

`&&` and `||` evaluate the left side first. If the left side alone determines the result, the right side is not evaluated at all.

> **Definition 7.8**
>
> **Short-circuit evaluation**: in the expression `a && b`, if a is `false`, the result is `false` whatever b is, and b is not evaluated. In `a || b`, if a is `true`, the result is `true` and b is not evaluated. The right side is evaluated only when it is needed to determine the result.

Let us trace the application program for three inputs:

```
int age = in.nextInt();
boolean hasDoc = in.nextBoolean();
boolean eligible = age >= 18 && hasDoc;
if (eligible) {
    System.out.println("Application accepted");
}
```

age	hasDoc	Left: `age >= 18`	Right: `hasDoc`	eligible	Output
20	true	true	read: true	true	Application accepted
20	false	true	read: false	false	(no output)
16	true	false	not read	false	(no output)

Table 7.10: Evaluation of the expression `eligible = age >= 18 && hasDoc` for three inputs.

In the third row the left side is false. Whatever the value of the right side, the result of the AND will be false; Java never looks at `hasDoc`. In this example, short-circuiting is only a small time saving. Its real importance appears where the right side can be evaluated safely only when the left side is true. A divisor that must not be zero in a division is such a condition:

```
if (b != 0 && a / b > 1) { … }
```

If b = 0, the left side is false and `a / b` is never evaluated. Division by zero with integers is an error that stops the program. Short-circuiting places the right side of the condition under the protection of the left side. Order matters: had `a / b > 1 && b != 0` been written, the division would be performed before the check.

7.5.3 De Morgan's laws

If we need to write the case in which the application is rejected, the most direct way is to put `!` in front of the acceptance condition: `!(age >= 18 && hasDoc)`. This form is correct but hard to read. The reader must first think about the inside of the parentheses and then about its opposite. De Morgan's laws let us distribute `!` into the parentheses and obtain a flatter condition.

Theorem 7.1 (De Morgan's laws)

For all boolean values a and b,

> !(a && b) == (!a || !b) and !(a || b) == (!a && !b)

hold. When `!` is distributed into the parentheses, `&&` and `||` swap places and each term is negated.

The proof is by truth table. There are only four combinations of values that a and b can take; if both sides give the same value in all four, the equality holds in every case. Table 7.11 gives the four rows. The table was produced by a Java program that computed the four combinations one by one.

a	b	`!(a && b)`	`!a || !b`	`!(a || b)`	`!a && !b`
false	false	true	true	true	true
false	true	true	true	false	false
true	false	true	true	false	false
true	true	false	false	false	false

Table 7.11: Truth table of De Morgan's laws. The third and fourth columns are the same in every row; so are the fifth and sixth columns.

When the rule is applied to the rejection condition, the term `!(age >= 18)` simplifies as well. “Not greater than or equal to 18” means “less than 18”. The rejection condition therefore takes this form:

```
// Reject: under 18 or no document
if (!(age >= 18 && hasDoc)) { … }
// Same condition by De Morgan:
if (age < 18 || !hasDoc) { … }
```

The equivalence of the two conditions was also tested with a program: age from 0 to 40 and both values of `hasDoc`, *41 × 2 = 82* combinations in all. In all 82 combinations the two conditions gave the same result. The opposites of the comparisons are in the last column of Table 7.2: `>=` ↔ `<`, `>` ↔ `<=`, `==` ↔ `!=`.

Example 7.1

How can the case in which a temperature is *not* between 15 and 30 (both bounds included) be written without using `!`?

Solution. The inside of the range is `t >= 15 && t <= 30`. The outside is its opposite: `!(t >= 15 && t <= 30)`. By De Morgan's law, `&&` turns into `||` and each term is negated: `!(t >= 15)` → `t < 15`, `!(t <= 30)` → `t > 30`. The result is `t < 15 || t > 30`. A temperature cannot be below 15 and above 30 at the same time, so if `&&` had been kept in the condition, no value would satisfy it.

7.5.4 The non-short-circuit & and |

In Java the single-character operators `&` and `|` can also be used with boolean values. Their logical results are the same as those of `&&` and `||`. The difference lies in the way of evaluation: the single-character operators *always* evaluate both sides. This difference becomes visible when the right side contains an error or a side effect.

```
public class ShortCircuit {
    public static void main(String[] args) {
        int a = 10, b = 0;
        if (b != 0 && a / b > 1) {
            System.out.println("big");
        }
        System.out.println("&& is safe");
        if (b != 0 &  a / b > 1) {
            System.out.println("big");
        }
        System.out.println("never reached");
    }
}
```

```
&& is safe
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ShortCircuit.main(ShortCircuit.java:8)
```

The line with `&&` passes safely and “&& is safe” is printed. In the line with `&`, however, `a / b` is evaluated even though b = 0. Java throws an **exception**: `ArithmeticException: / by zero`. The program stops at line 8, and the last `println` is never reached.

One way to see the side effect is to write a small method that increments a counter every time it is called:

```
static int count = 0;
static boolean check(boolean v) { count++; return v; }
```

Expression	Result	count	Explanation
`check(false) && check(true)`	false	1	Left is false; right is not called.
`check(false) & check(true)`	false	2	Both sides are called.
`check(true) || check(true)`	true	1	Left is true; right is not called.
`check(true) | check(true)`	true	2	Both sides are called.

Table 7.12: Before each expression, count is set to 0. While the logical result stays the same, the counter shows how many sides were evaluated.

The values in the table are the output of the program below. In each line, the result is printed first and then the counter.

```
public class Counter {
    static int count = 0;
    static boolean check(boolean v) { count++; return v; }

    public static void main(String[] args) {
        boolean r;
        count = 0; r = check(false) && check(true); System.out.println(r + " " + count);
        count = 0; r = check(false) &  check(true); System.out.println(r + " " + count);
        count = 0; r = check(true)  || check(true); System.out.println(r + " " + count);
        count = 0; r = check(true)  |  check(true); System.out.println(r + " " + count);
    }
}
```

```
false 1
false 2
true 1
true 2
```

Conditions almost always use `&&` and `||`. The real use of `&` and `|` is bit operations on integers.

Common mistake

Putting an operation that changes a variable (such as `++k`) inside a condition. Because of short-circuiting, whether this operation runs depends on the value of the left side. In the line `if (k > 0 && ++k > 5)`, k is incremented only when it is positive. The behavior of such code is hard to follow; the increment should be done on a separate line before the condition.

7.5.5 Operator precedence

A condition can contain arithmetic, comparison, and logical operators together. **Operator precedence** determines which is evaluated first. The precedence of the operators that occur often in conditions is given in Table 7.13, from highest to lowest.

Rank	Operators	Kind
1	`!`, unary `-`, `++`, `--`	unary
2	`*` `/` `%`	multiplicative
3	`+` `-`	additive
4	`<` `<=` `>` `>=`	relational
5	`==` `!=`	equality
6	`&`	non-short-circuit AND
7	`|`	non-short-circuit OR
8	`&&`	AND
9	`||`	OR
10	`? :`	conditional operator
11	`=` `+=` ...	assignment

Table 7.13: Precedence of operators that occur often in conditions (the upper one is evaluated first).

Thanks to this ordering, `year % 4 == 0 && year % 100 != 0` can be written without parentheses. First the `%` operations are evaluated, then the `==` and `!=` comparisons, and `&&` last. It is also important that `&&` comes before `||`: the expression `a || b && c` is read as `a || (b && c)`. The precedence of `!` is the highest, so parentheses are needed when negating a comparison: `!(x > 5)`. Without parentheses, `!x > 5` means `(!x) > 5` and does not compile if x is a number. Whenever you are unsure about precedence, using parentheses both prevents errors and helps the reader.

7.6 The conditional operator ?:

When the only job of a decision is to give a variable one of two values, if-else takes four lines:

```
int max;
if (a > b) max = a;
else       max = b;
```

Java offers a shorter notation for this pattern.

Definition 7.9

The **conditional operator** (also called the ternary operator because it takes three operands) is written as `condition ? value1 : value2`. If the condition is `true`, the value of the expression is value1; if it is `false`, the value is value2. Only the selected value is evaluated.

An if-else is a statement: it does something but does not produce a value. The conditional operator, however, is an **expression**: it produces a value. It can therefore be used on the right side of an assignment, in the argument of a method, or inside another expression.

```
int a = 7, b = 12;
int max = (a > b) ? a : b;
System.out.println(max);
```

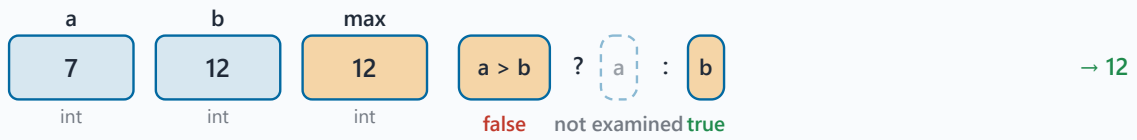


Figure 7.11: Evaluation of the expression `(a > b) ? a : b` for a = 7, b = 12. Because the condition is false, the b after the colon is chosen; a is not examined. max = 12.

The steps are as follows. First the condition is evaluated: *7 > 12* is false. Because the condition is false, the value after the colon (`:`), that is, b, is chosen. The value of the expression is 12 and it is assigned to `max`. The program prints 12:

```
12
```

The conditional operator can also be nested. An expression that finds the sign of a number as "+", "-", or "0" looks like this:

```
String sign = n > 0 ? "+" : n < 0 ? "-" : "0";
```

The conditional operator groups from the right. The expression is read as `n > 0 ? "+" : (n < 0 ? "-" : "0")`. For n = −5, the first condition is false; the part in parentheses is evaluated, and since *−5 < 0* is true, "-" is chosen. This is a short and correct expression, but three or four levels of nested conditional operators become unreadable. The conditional operator is most useful for short selections between two values. When the options multiply, an else-if chain or a switch is clearer.

The two values of a conditional operator must have compatible types. `int m = a < b ? a : "none";` does not compile, because one branch of the expression is an int and the other is a string, and the result cannot be put into an int variable.

7.7 Comparing strings: == and equals

Numbers are compared with `==`. Strings (`String`) are different, and this is one of the points where Java beginners most often get stuck. A string variable stores not the text itself but the location of the object that holds the text in memory, that is, a **reference**. `==` compares two references: do the two variables point to *the same object*? The `equals` method is used to compare content.

```
String s1 = "java";
String s2 = "java";
String s3 = new String("java");
System.out.println(s1 == s2);
System.out.println(s1 == s3);
System.out.println(s1.equals(s3));
```

```
true
false
true
```

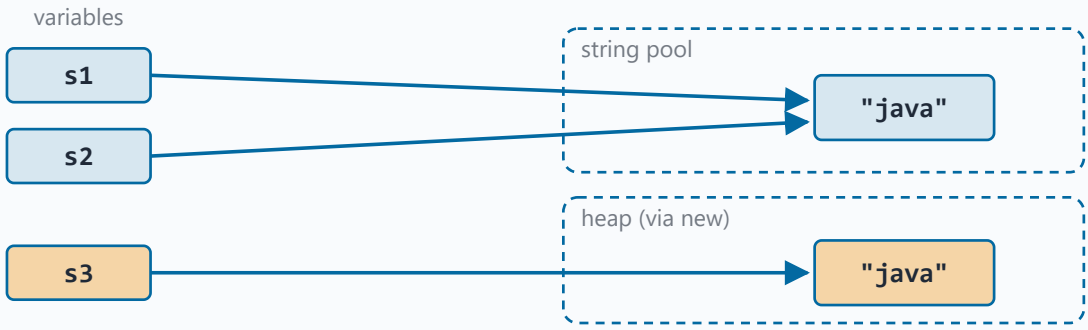


Figure 7.12: s1 and s2 point to the same "java" object in the string pool. s3 points to a separate "java" object created on the heap with `new`. The two objects have the same content but different locations.

Figure 7.12 shows the state of the three variables in memory. A piece of text written in quotation marks in code is called a **string literal**. Java keeps string literals in a shared region called the **string pool**. When the same literal is written a second time, no new object is created; the object in the pool is shared. That is why s1 and s2 point to the same object. `new String("java")`, however, always creates a new object. This object is in the **heap**, the region where objects created with `new` are kept.

Expression	What does it compare?	Result	Why?
`s1 == s2`	reference	true	Both point to the same object in the pool.
`s1 == s3`	reference	false	s3 points to the separate object on the heap.
`s1.equals(s3)`	content	true	The two objects have the same characters: j, a, v, a.

Table 7.14: Explanation of the three comparisons.

The first line giving `true` is dangerous, because it gives the impression that `==` compares content. This result is only a by-product of the pool. A string read from the user does not enter the pool; `Scanner` creates it as a new object. The following program shows this:

```
import java.util.Scanner;

public class StringInput {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String answer = in.next();
        System.out.println(answer == "yes");
        System.out.println(answer.equals("yes"));
        System.out.println(answer.equalsIgnoreCase("YES"));
    }
}
```

When the user types “yes”, the output is:

```
false
true
true
```

Even though the user typed exactly “yes”, `answer == "yes"` is false: one is a new object read from the keyboard, the other is the constant in the pool. `equals` gives the correct result. When the user types “Yes”, however, `equals` also gives false, because it is case-sensitive: “Y” and “y” are different characters. A comparison that ignores letter case is done with `equalsIgnoreCase`:

```
false
false
true
```

Common mistake

Comparing strings with `==`. In small experiments with literals the program seems to work correctly; with strings read from the user or concatenated inside the program, the result comes out wrong. The rule is simple: always compare strings with `equals`. If the string

variable can be `null`, put the literal first: `"yes".equals(answer)`. This form does not throw an error when answer is `null`; it simply returns `false`.

7.8 switch

An else-if chain can build any multiway decision. In some decisions, however, all the conditions test the same variable in the same way: `month == 1`, `month == 2`, … `month == 12`. Grouping the twelve months into seasons would need a long `||` chain. The `switch` construct makes decisions of this kind shorter and more readable.

7.8.1 When to use switch

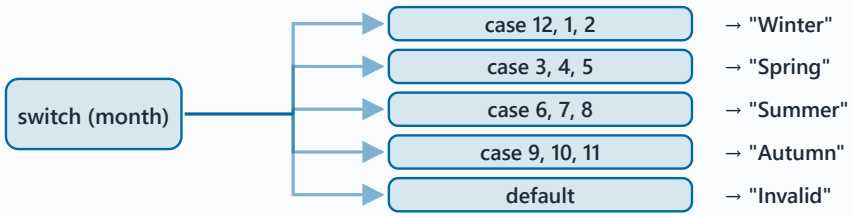


Figure 7.13: A switch expression goes directly to the matching case according to the value of month. Each case lists one or several constant values; if none matches, default is chosen.

`switch` compares the value of a single expression with a series of constants and goes to the matching option. In Figure 7.13, the month value is routed to one of five options. The options are written with **case labels**. Each label carries one or several constant values. If no label matches, `default` is chosen.

In an else-if chain, the conditions are tried in order. In a `switch`, the program goes directly to the case that matches the value; the labels in between are not “tried and passed”. A switch is therefore suitable for choices made among the discrete values of a single variable: a month number, a day number, a menu option, an operator sign. Range conditions (“if the score is greater than 85”) and complex conditions that depend on several variables are written with if-else.

The value in a case label must be a **constant** known at compile time: `4`, `'A'`, `"+"`. A variable cannot be a label:

```
public class VariableCase {
    public static void main(String[] args) {
        int month = 4, limit = 4;
        switch (month) {
            case limit -> System.out.println("equal");
            default -> System.out.println("different");
        }
    }
}
```

```
VariableCase.java:5: error: constant expression required
            case limit -> System.out.println("equal");
                 ^
1 error
```

The expression that a switch tests can be of types such as `int`, `char`, and `String`. `double` and `boolean` values cannot be tested with a switch; use if-else for them.

7.8.2 The switch expression

In the newer notation introduced in Java 14, each case is written with an arrow (`->`) and a switch can produce a value.

Definition 7.10

A **switch expression** is a switch that produces the value of the selected case as its result: `x = switch (…) { … };`. A **switch statement**, on the other hand, does not produce a value; it only runs the code of the selected case. A case written with an arrow is said to use the **arrow form**; in the arrow form, only the selected case runs.

```
String season = switch (month) {
    case 12, 1, 2 -> "Winter";
    case 3, 4, 5 -> "Spring";
    case 6, 7, 8 -> "Summer";
    case 9, 10, 11 -> "Autumn";
    default -> "Invalid";
};
```

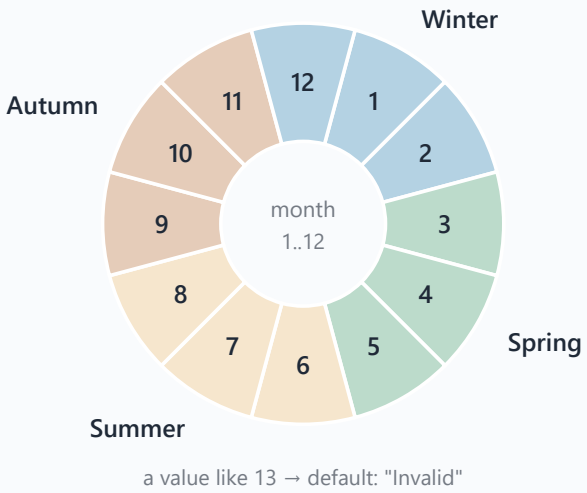


Figure 7.14: The season wheel. 12, 1, and 2 are Winter; 3, 4, and 5 are Spring; 6, 7, and 8 are Summer; 9, 10, and 11 are Autumn. Values outside 1–12 go to default, “Invalid”.

Each line in the code is a case. The line `case 12, 1, 2 -> "Winter";` means “if month is 12, 1, or 2, the value is "Winter"”. A case can list several values separated by commas. In this way the twelve months are covered in five lines (Figure 7.14). The switch expression as a whole is a value; this value is assigned to the variable `season`. Because this is an assignment statement, a semicolon follows the closing brace: `};`.

A switch expression must produce a result for every possible value. An `int` can take more than 4 billion values; the cases list only twelve of them. That is why `default` is mandatory. If it is deleted, the Java program does not compile:

```
NoDefault.java:4: error: the switch expression does not cover all possible input values
        String season = switch (month) {
                        ^
1 error
```

The message means “the switch expression does not cover all possible input values”. In a switch statement, however, `default` is optional; if no case matches, the switch ends without doing anything. The compiler also reports an error if the same value appears in two cases, because it would be unclear which branch the value goes to:

```
DuplicateCase.java:6: error: duplicate case label
            case 2, 3, 4, 5 -> "Spring";
            ^
1 error
```

7.8.3 Tracing a switch expression step by step

```
int month = in.nextInt();
String season = switch (month) {
    case 12, 1, 2 -> "Winter";
    case 3, 4, 5 -> "Spring";
    case 6, 7, 8 -> "Summer";
    case 9, 10, 11 -> "Autumn";
    default -> "Invalid";
};
System.out.println(season);
```

month	Matching case	Value of the switch	season	Output
4	`case 3, 4, 5`	"Spring"	"Spring"	Spring
12	`case 12, 1, 2`	"Winter"	"Winter"	Winter
13	none → `default`	"Invalid"	"Invalid"	Invalid

Table 7.15: Three runs of the season program.

For month = 4, the program goes to the line `case 3, 4, 5`, where 4 is in the list. The value of this case is "Spring". It is assigned to `season` as the value of the switch expression and printed. The code of no other case runs. month = 13 is in no list; `default` is chosen. The program was also tried for 1, 7, 10, 0, and −3: it printed Winter, Summer, Autumn, Invalid, and Invalid.

```
Spring
```

```
Winter
```

```
Invalid
```

7.8.4 A branch with a block, and yield

The right side of a case in the arrow form is a single expression. Sometimes a branch has to do other work before it produces the value: print a message, compute an intermediate value. Such a branch is put into a block with braces, and the value is given with the keyword `yield`.

Definition 7.11

yield is the statement that specifies, in a block branch of a switch expression, the value that the branch produces. When `yield` runs, the block ends and the given value becomes the value of the switch expression.

```
int year = 2024, month = 2;
int days = switch (month) {
    case 4, 6, 9, 11 -> 30;
    case 2 -> {
        System.out.println("February");
        yield year % 4 == 0 ? 29 : 28;
    }
    default -> 31;
};
System.out.println(days);
```

```
February
29
```

Because month = 2, the branch with the block is selected. The block first prints “February”. Then the `yield` line runs. The value here is computed with a conditional operator: since *2024 % 4 = 0*, `year % 4 == 0` is true and 29 is chosen. The value of the switch expression is 29; it is assigned to `days` and printed. The output is two lines: “February” and “29”.

The same program was also run with other inputs. For February of 2023, it printed “February” and 28 (*2023 % 4 = 3*). For April it printed only 30, and for January only 31; since the block was not selected in these two runs, the “February” line was not printed either. In this example the leap year rule is simplified: it checks only divisibility by 4. The program therefore also prints 29 for 1900, although 1900 is not a leap year. The full rule is given in the leap year example.

If a branch with a block ends without `yield`, the branch has no value. Java catches this at compile time:

```
public class NoYield {
    public static void main(String[] args) {
        int year = 2024, month = 2;
        int days = switch (month) {
            case 4, 6, 9, 11 -> 30;
            case 2 -> {
                System.out.println("February");
            }
            default -> 31;
        };
        System.out.println(days);
    }
}
```

```
NoYield.java:8: error: switch rule completes without providing a value
            }
            ^
  (switch rules in switch expressions must either provide a value or throw)
1 error
```

The message means “the switch rule completes without providing a value”; the extra line reminds us that rules in switch expressions must either provide a value or throw an exception.

7.8.5 Classic switch, break, and fall-through

Before the arrow form, a switch in Java was written only with a colon (`:`). This classic notation is still valid today and is often seen in old code. In a classic switch, case labels are only entry points. The program enters at the matching label and runs all the code below until it sees `break` or reaches the end of the switch.

Definition 7.12

Fall-through is what happens in a classic switch when the code of a case finishes without a `break`: execution passes into the code of the next case. The label of the next case is not checked in the process. The `break` statement exits the switch.

```
int day = 2;
switch (day) {
  case 1: System.out.println("Mon"); break;
  case 2: System.out.println("Tue");
  case 3: System.out.println("Wed"); break;
  default: System.out.println("None");
}
```

```
Tue
Wed
```

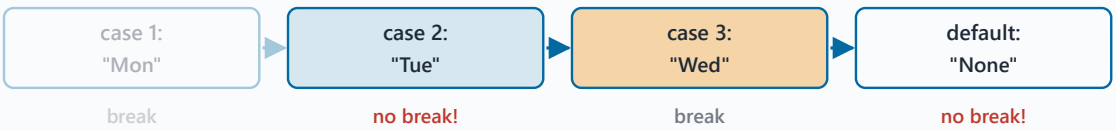


Figure 7.15: The classic switch for day = 2. Execution enters case 2 directly and prints “Tue”. Because case 2 has no break, it falls through into the code of case 3 and prints “Wed”. The break in case 3 exits the switch; default does not run.

The steps are as follows. day = 2. The program goes directly to the label `case 2:`; case 1 is not checked. “Tue” is printed. There is no `break` on this line. Execution passes to the line below, the code of `case 3:`. That the label is 3 does not matter; a label is only an entry point. “Wed” is printed. The `break` on this line exits the switch and `default` does not run (Figure 7.15). The same program printed only “Mon” for day = 1, only “Wed” for 3, and “None” for 9.

Fall-through is sometimes used on purpose: so that several cases do the same job, the labels are written one under another and the code is put only under the last one. In the arrow form, this need is met by the comma-separated list. The following program separates weekdays from weekend days without needing fall-through:

```
import java.util.Scanner;

public class DayType {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int day = in.nextInt();
        switch (day) {
            case 1, 2, 3, 4, 5 -> System.out.println("Weekday");
            case 6, 7 -> System.out.println("Weekend");
            default -> System.out.println("Invalid day");
        }
    }
}
```

```
Weekday
```

```
Weekend
```

```
Invalid day
```

Fall-through never happens in the arrow form: the code of the selected case runs and the switch ends. There is no need to write `break`. Using the arrow form in new code prevents at the outset the error caused by a forgotten `break`.

Common mistake

Forgetting `break` in a classic switch. The program compiles and may even work correctly for most inputs; the error shows up only when the case missing its `break` is selected, through the extra lines it prints. Another misconception is to think that `default` always runs last. `default` can also be written in the middle of a switch, and the fall-through rule applies to it too: if it has no `break`, execution falls into the code of the case below it.

7.8.6 if-else or switch?



Figure 7.16: Left, number ranges with if-else: below 50 Fail, 50–85 Pass, 85 and above Excellent. Right, discrete values of a single variable with switch: 1–5 weekday, 6–7 weekend.

Figure 7.16 puts the natural areas of use of the two constructs side by side. Score ranges are continuous: between 50 and 85 there are infinitely many decimal values, and it is impossible to list them all as cases. Days, on the other hand, are discrete: there are only seven values and each can be listed by name. Table 7.16 compares the two constructs by several criteria.

Criterion	if-else	switch
Range condition	Suitable	Not directly suitable
Many options by a single value	Usable	Usually clearer
Complex logic	Suitable	Limited
Each branch produces a result	Needs an assignment	A switch expression can return a value

Table 7.16: Comparison of if-else and switch.

The last row shows an important advantage. In an else-if chain, every branch assigns to the same variable separately, and it is easy to forget the assignment in one branch. In a switch expression, the value is assigned in a single place and the compiler checks that every possible value has a result.

7.9 Examples

The examples in this section combine decision structures in realistic problems. Each example gives the problem and the idea of the solution first, then the code, then a trace for a few inputs and the program's actual output. In most examples the code itself is short; the real work is finding which inputs behave specially.

7.9.1 Odd or even?

An integer is even if it is divisible by 2 with no remainder. In Java the remainder is found with the `%` operator. The **remainder** is the amount left over after a division: dividing 17 by 2 gives quotient 8 and remainder 1, because *17 = 2 × 8 + 1*.

```
int num = in.nextInt();
if (num % 2 == 0)
    System.out.println("Even");
else
    System.out.println("Odd");
```

In the condition, `%` is evaluated first and then the result is compared with 0; `%` has higher precedence than `==`. Let us trace the program for 17, −4, and −7.

num	num % 2	num % 2 == 0	Output
17	1	false	Odd
−4	0	true	Even
−7	−1	false	Odd

Table 7.17: Three runs of the odd/even program.

```
Odd
```

```
Even
```

```
Odd
```

The third row needs attention: the value of `-7 % 2` is −1, not 1. In Java, integer division rounds toward zero: *−7 / 2 = −3*, not −4. The remainder is computed consistently with this quotient: *−7 = 2 × (−3) + (−1)*. As a result, the sign of the remainder is always the same as the sign of the dividend. Figure 7.17 shows the remainders modulo 2 of the numbers from −4 to 4.

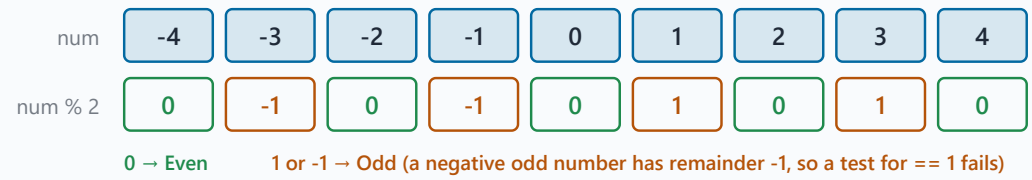


Figure 7.17: The values of `num % 2` for the numbers from −4 to 4. For even numbers the remainder is 0. For positive odd numbers the remainder is 1, and for negative odd numbers it is −1.

The values in the figure were also produced by a program. The second line prints the two conditions for −7, the quotient `-7 / 2`, and `7 % -2`:

```
public class RemainderTable {
    public static void main(String[] args) {
        for (int num = -4; num <= 4; num++) {
            System.out.print(num % 2 + " ");
        }
        System.out.println();
        System.out.println((-7 % 2 == 1) + " " + (-7 % 2 != 0) + " " + (-7 / 2) + " " + (7
% -2));
    }
}
```

```
0 -1 0 -1 0 1 0 1 0
false true -3 1
```

`-7 % 2 == 1` is false and `-7 % 2 != 0` is true. That `7 % -2` comes out as 1 also confirms the rule: even if the divisor is negative, the remainder takes the sign of the dividend, 7.

Common mistake

Testing for an odd number with the condition `num % 2 == 1`. This condition works for positive odd numbers but gives the wrong result for negative odd numbers: −7 is counted as "not odd". Safe conditions compare the remainder with 0: `num % 2 == 0` for even and `num % 2 != 0` for odd. These two conditions are correct for every int.

7.9.2 Absolute value and overflow

The **absolute value** of a number is its distance from zero: *|−8| = 8*, *|5| = 5*. If the number is negative its sign is changed; otherwise it is left as it is. This is the typical example of the one-way `if`: for non-negative numbers there is nothing to do.

```
int num = in.nextInt();
if (num < 0) {
    num = -num;
}
System.out.println(num);
```

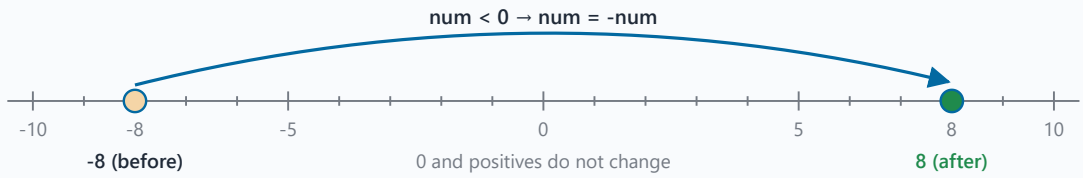


Figure 7.18: Absolute value on the number line. −8 is reflected about zero and becomes 8. 0 and positive values do not change.

Input	`num < 0`	`num = -num`	Output
−8	true	*−(−8) = 8*	8
5	false	skipped	5
−2147483648	true	*−(−2147483648)* → overflow → −2147483648	−2147483648

Table 7.18: Three runs of the absolute value program. The third input is Integer.MIN_VALUE.

```
8
```

```
5
```

```
-2147483648
```

The first two rows give the expected result. The result in the third row, however, is surprising: the absolute value of a number has come out negative. The cause is the limits of the int type. An int is stored in 32 bits and can take values between $-2^{31} = -2{,}147{,}483{,}648$ and $2^{31} - 1 = 2{,}147{,}483{,}647$. The positive counterpart of the smallest value, `Integer.MIN_VALUE`, is 2,147,483,648. This number is one more than the largest int and does not fit in an int.

Definition 7.13

Overflow is when the result of an operation falls outside the range of values that the type storing the result can hold. In Java, overflow in int and long operations does not cause an error; the result "wraps around" to the other end of the range. Adding 1 to the largest int gives the smallest int.

The correct result of *−(−2,147,483,648)* is 2,147,483,648. This value exceeds the largest int by 1 and wraps around to the other end of the range: the result is again −2,147,483,648. Java's built-in `Math.abs` method behaves the same way. There are two ways to handle this special case correctly. The first is to do the computation in a wider type, long. The second is to use the `Math.absExact` method, available since Java 15; instead of silently giving a wrong result, this method throws an exception.

```
public class AbsEdgeCase {
    public static void main(String[] args) {
        int num = Integer.MIN_VALUE;
        System.out.println(Integer.MAX_VALUE);
        System.out.println(Math.abs(num));
        long wide = num;
        if (wide < 0) wide = -wide;
        System.out.println(wide);
        System.out.println(Math.absExact(num));
    }
}
```

```
2147483647
-2147483648
2147483648
Exception in thread "main" java.lang.ArithmeticException: Overflow to represent absolute
value of Integer.MIN_VALUE
        at java.base/java.lang.Math.absExact(Math.java:1903)
        at AbsEdgeCase.main(AbsEdgeCase.java:9)
```

The first line is the largest int. The second line shows that `Math.abs` also gives a negative result. In the third line the value has been copied into a long; the range of long is much wider and the correct result, 2147483648, is obtained. In the last line `Math.absExact` notices the overflow and throws an `ArithmeticException`. The message means that representing the absolute value of Integer.MIN_VALUE would overflow.

7.9.3 Leap year

The Earth takes about 365.2425 days to complete one orbit around the Sun. The calendar balances this surplus with leap years, that is, years of 366 days. The rule of the Gregorian calendar has three steps:

1. years divisible by 4 are leap years;
2. but years divisible by 100 are not leap years;
3. but years divisible by 400 are leap years again.

The rule can be collected into a single boolean expression: “divisible by 4 and not divisible by 100” or “divisible by 400”.

```
int year = in.nextInt();
boolean leap =
    (year % 4 == 0 && year % 100 != 0)
    || year % 400 == 0;
if (leap)
    System.out.println(year + " is a leap year");
else
    System.out.println(year + " is not a leap year");
```

The parentheses show that the part with `&&` is a unit. Since `&&` has higher precedence than `||`, the result would not change without the parentheses, but the parentheses show the structure of the rule clearly to the reader. Let us trace the program for four years.

year	`year % 4 == 0`	`year % 100 != 0`	**Parentheses**	`year % 400 == 0`	**leap**
2024	0 → true	24 → true	true	not examined	true
2023	3 → false	not examined	false	23 → false	false
1900	0 → true	0 → false	false	300 → false	false
2000	0 → true	0 → false	false	0 → true	true

Table 7.19: Evaluation of the leap year condition for four years. “not examined” is a term that is not evaluated because of short-circuiting.

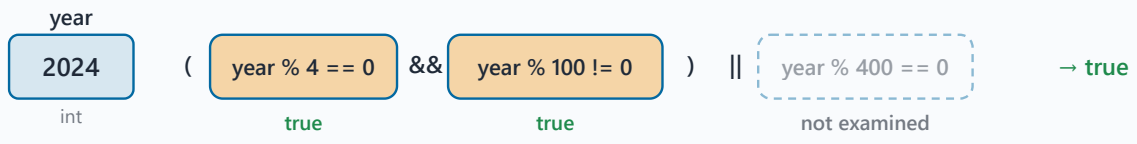


Figure 7.19: Evaluation of the condition for year = 2024. Both terms inside the parentheses are true. Because the left side is true, `||` short-circuits; the term `year % 400 == 0` is not examined.

In each row there are two chances for short-circuiting. In 2024 the inside of the parentheses is true and the right side of `||` is not examined (Figure 7.19). In 2023 the first term is false; `&&` does not evaluate the second term. Because the inside of the parentheses is false, this time the right side of `||` is evaluated. 1900 and 2000 are the subtle points of the rule. Both are divisible by 100 and the inside of the parentheses comes out false. The difference is in the last term: *1900 % 400 = 300*, *2000 % 400 = 0*. That is why 2000 is a leap year and 1900 is not. The program printed "is not a leap year" for 2100 and "is a leap year" for 1600.

```
2024 is a leap year
```

```
2023 is not a leap year
```

```
1900 is not a leap year
```

```
2000 is a leap year
```

7.9.4 The quadratic equation

The number of real roots of the equation $ax^2 + bx + c = 0$ depends on the discriminant: $\Delta = b^2 - 4ac$. If Δ is positive there are two distinct real roots; if it is zero there is a single (repeated) root; if it is negative there are no real roots. The roots are found with this formula:

$$x_{1,2} = (-b \pm \sqrt{\Delta}) / (2a)$$

The three cases are separated by an else-if chain with three branches. Here a is assumed to be nonzero; if a = 0, the equation would not be quadratic.

```
double a = in.nextDouble();
double b = in.nextDouble();
double c = in.nextDouble();
double d = b * b - 4 * a * c;
if (d > 0) {
  double x1 = (-b + Math.sqrt(d)) / (2 * a);
  double x2 = (-b - Math.sqrt(d)) / (2 * a);
  System.out.println(x1 + " and " + x2);
} else if (d == 0) {
  System.out.println(-b / (2 * a));
} else {
  System.out.println("No real roots");
}
```

a, b, c	d = b² − 4ac	Branch executed	Computation	Output
1.0, −3.0, 2.0	*9.0 − 8.0 = 1.0*	`d > 0`	*x1 = (3.0 + 1.0) / 2.0 = 2.0, x2 = (3.0 − 1.0) / 2.0 = 1.0*	2.0 and 1.0
1.0, −2.0, 1.0	*4.0 − 4.0 = 0.0*	`d == 0`	*−b / 2a = 2.0 / 2.0 = 1.0*	1.0
1.0, 0.0, 1.0	*0.0 − 4.0 = −4.0*	`else`	—	No real roots

Table 7.20: Three runs of the equation program. A different branch runs in each row.

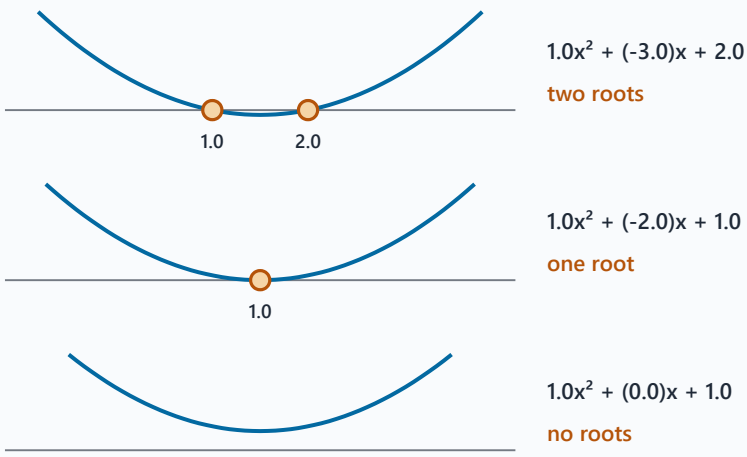


Figure 7.20: The parabolas of the three runs. (a) Δ = 1.0: the parabola crosses the x-axis at 1.0 and 2.0. (b) Δ = 0.0: the parabola is tangent to the axis at 1.0. (c) Δ = −4.0: the parabola does not cross the axis.

```
2.0 and 1.0
```

```
1.0
```

```
No real roots
```

Figure 7.20 gives the geometric meaning of each run. The program was also tried for $2x^2 - 7x + 3 = 0$: $\Delta = 49 - 24 = 25$, $\sqrt{\Delta} = 5$; the roots are $(7 + 5) / 4 = 3.0$ and $(7 - 5) / 4 = 0.5$. The program printed “3.0 and 0.5”.

The `else` branch does more than print a message; it also protects the program from an error. The square root of a negative number is undefined in the real numbers. In Java, `Math.sqrt(-4.0)` does not throw an exception; it returns `NaN` (Not a Number). Division by zero with floating-point numbers does not cause an error either; it produces infinity (`Infinity`):

```
public class FloatEdgeCases {
    public static void main(String[] args) {
        System.out.println(Math.sqrt(-4.0));
        System.out.println(7.0 / 0);
        System.out.println(-7.0 / 0);
        System.out.println(0.0 / 0);
        System.out.println(0.1 + 0.2 == 0.3);
        System.out.println(0.1 + 0.2);
    }
}
```

```
NaN
Infinity
-Infinity
NaN
false
0.30000000000000004
```

Every computation that involves `NaN` again gives `NaN`, and the error goes unnoticed until the results are printed on the screen. That is why the order of decisions matters: `Math.sqrt(d)` is called only in the branch where *d > 0* is known.

The last two lines show another subtlety of floating-point numbers. `0.1 + 0.2` is not exactly 0.3; 0.1 and 0.2 cannot be represented exactly in binary, and a small rounding difference remains. The comparison `d == 0` is safe in this example because the inputs are integers and $b^2 - 4ac$ is computed exactly. For floating-point values that come from measurement, however, a small tolerance is used instead of `==`: `Math.abs(d) < 1e-9`.

7.9.5 The largest of three numbers

One way to find the largest of three numbers is to write every possibility as a separate condition: “if a is greater than b and than c …”. This approach gets complicated even with three numbers and becomes unmanageable with five. A better way is the “candidate and compare” pattern: the first number is taken as the candidate, then each number is compared with the candidate, and if it is larger than the candidate it becomes the candidate.

```
int a = in.nextInt();
int b = in.nextInt();
int c = in.nextInt();
int max = a;
if (b > max)
    max = b;
if (c > max)
    max = c;
System.out.println("Max: " + max);
```

Input (a, b, c)	Step	Conditions	max
7, 12, 5	`max = a`	—	7
	`b > max`	*12 > 7* → true	12
	`c > max`	*5 > 12* → false	12
3, 3, 9	`max = a`	—	3
	`b > max`	*3 > 3* → false	3
	`c > max`	*9 > 3* → true	9
8, 2, 4	`max = a`	—	8
	`b > max`	*2 > 8* → false	8
	`c > max`	*4 > 8* → false	8

Table 7.21: Three runs of the “candidate and compare” pattern. The last column is the candidate value after each step.

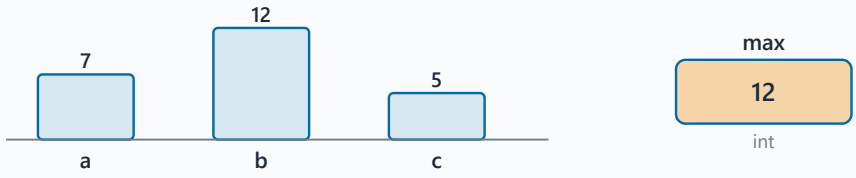


Figure 7.21: The state after the comparison with b for the input (7, 12, 5). The bars show the sizes of a, b, and c; the candidate has become max = 12.

In the second input, *3 > 3* is false; an equal value does not change the candidate. The result is still correct, because it does not matter which of the equal values is the candidate. The strength of the pattern is that it generalizes: for n numbers exactly n − 1 comparisons are needed, and each comparison has the same form. When this pattern is put in the body of a loop, it finds the largest of any number of numbers. The same job can also be done in one line with Java's built-in method: `Math.max(a, Math.max(b, c))`. The complete program prints the results of both methods:

```
import java.util.Scanner;

public class Largest {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int a = in.nextInt();
        int b = in.nextInt();
        int c = in.nextInt();
        int max = a;
        if (b > max)
            max = b;
        if (c > max)
            max = c;
        System.out.println("Max: " + max);
        System.out.println(Math.max(a, Math.max(b, c)));
    }
}
```

```
Max: 12
12
```

```
Max: -1
-1
```

Common mistake

Starting the candidate with 0 instead of the first number: `int max = 0;`. When the inputs are positive, this also gives the right result. But with the input (−2, −7, −1), no number is greater than 0; the candidate never changes and 0, which is not even one of the inputs, is printed. The correct program prints −1 for this input. The candidate is always initialized with one of the inputs.

7.9.6 A calculator with switch

Let the user enter two numbers with an operator sign between them: `8.0 * 2.5`. The program reads the sign as a string and chooses the corresponding operation with a switch. The `next()` method of `Scanner` reads the word up to the next whitespace as a string; that is why the input has spaces between the numbers and the sign.

```
double x = in.nextDouble();
String op = in.next();
double y = in.nextDouble();
switch (op) {
  case "+" -> System.out.println(x + y);
  case "-" -> System.out.println(x - y);
  case "*" -> System.out.println(x * y);
  case "/" -> {
    if (y == 0) System.out.println("Error!");
    else System.out.println(x / y);
  }
  default -> System.out.println("Invalid");
}
```

Input	op	Matching case	Extra condition	Output
8.0 * 2.5	"*"	`case "*"`	—	20.0
7.0 / 0.0	"/"	`case "/"`	`y == 0` → true	Error!
5.0 % 3.0	"%"	none → `default`	—	Invalid

Table 7.22: Three runs of the calculator.



Figure 7.22: For the input 8.0 * 2.5, the switch goes to the case that carries op = "*" and prints 20.0.

```
20.0
```

```
Error!
```

```
Invalid
```

The division branch is not a single expression but a block containing an if-else. This is a switch statement and does not produce a value; therefore `yield` is not needed in the block. The check in the block is necessary, because division by zero with floating-point numbers does not cause an error: the expression `7.0 / 0` produces `Infinity`. To give the user a clear message instead of printing

“Infinity”, the divisor is tested before the division. In the third run the “%” sign matches no case; `default` catches the unexpected input. A switch on a String matches with `equals`; the reference problem caused by using `==` on strings does not arise here.

The program was also tried for `9 - 4`, `7 / 2`, and `3 + 4.5`; it printed 5.0, 3.5, and 7.5. Because x and y are double, `7 / 2` is not integer division and gives 3.5.

7.9.7 Worked examples from the exercises

The following examples are selected from the exercises of the slides. Each applies a pattern described in this chapter.

Example 7.2

A program that prints whether the integer entered is positive, negative, or zero.

Solution. The three options are mutually exclusive; a number falls into only one of them. That is a job for an else-if chain. No condition is needed for the third case: if the number is neither positive nor negative, it is zero.

```
import java.util.Scanner;

public class SignCheck {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int num = in.nextInt();
        if (num > 0)
            System.out.println("Positive");
        else if (num < 0)
            System.out.println("Negative");
        else
            System.out.println("Zero");
    }
}
```

It prints “Positive” for 12, “Negative” for −3, and “Zero” for 0. 0 is a boundary value and was tested separately: *0 > 0* and *0 < 0* are false, so the else runs.

Example 7.3

A program that computes the body mass index from height (in meters) and weight in kilograms, *BMI = weight / height²*, and prints its category. The limits: below 18.5 is underweight, from 18.5 to 25 is normal, and 25 and above is overweight.

Solution. The categories divide a continuous range; switch is not suitable. Because the thresholds are ordered from smallest to largest, the conditions are written with `<` and the chain starts from the smallest threshold.

```
import java.util.Scanner;

public class Bmi {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        double height = in.nextDouble();
        double weight = in.nextDouble();
        double bmi = weight / (height * height);
        if (bmi < 18.5)
            System.out.println("Underweight");
        else if (bmi < 25)
            System.out.println("Normal");
        else
            System.out.println("Overweight");
    }
}
```

For a height of 1.75 m and a weight of 70 kg, *BMI = 70 / 3.0625 ≈ 22.86* and “Normal” is printed. For 1.60 m and 45 kg, *45 / 2.56 ≈ 17.58* and “Underweight” is printed; for 1.80 m and 90 kg, *90 / 3.24 ≈ 27.78* and “Overweight” is printed. The input to the program is given with a decimal point, as in Java source code: 1.75. If the computer's regional setting is Turkish, `Scanner` may expect a decimal comma; in that case type 1,75.

Example 7.4

Do three side lengths form a triangle? If so, is the triangle equilateral, isosceles, or scalene?

Solution. For three sides to form a triangle, the sum of any two sides must be greater than the third (the triangle inequality). First the opposite of this condition is checked: if any sum is equal to or smaller than the third side, there is no triangle. By De Morgan's law, the opposite of “all three inequalities are true” is “at least one is false”; that is why the terms are joined with `||`. Then the type of triangle is tested from the narrow condition to the broad one: first equality of all three sides, then equality of any two sides.

```
import java.util.Scanner;

public class Triangle {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int a = in.nextInt(), b = in.nextInt(), c = in.nextInt();
        if (a + b <= c || a + c <= b || b + c <= a)
            System.out.println("Not a triangle");
        else if (a == b && b == c)
            System.out.println("Equilateral");
        else if (a == b || b == c || a == c)
            System.out.println("Isosceles");
        else
            System.out.println("Scalene");
    }
}
```

The program printed Equilateral for (3, 3, 3), Isosceles for (3, 3, 5) and (5, 3, 5), and Scalene for (3, 4, 5). For (1, 2, 3), since *1 + 2 = 3*, it printed “Not a triangle”; these sides form a straight line. For (2, 9, 4), since *2 + 4 = 6 < 9*, the same result came out. The equilateral condition must be asked before the isosceles one, because every equilateral triangle also satisfies the isosceles condition.

Example 7.5

A program that converts a day number from 1 to 7 into a day name and warns on invalid input.

Solution. A single integer has seven discrete values; a switch expression is made for exactly this job. Invalid input is caught by `default`. Because this is a switch expression, `default` is mandatory anyway.

```
import java.util.Scanner;

public class DayName {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int day = in.nextInt();
        String name = switch (day) {
            case 1 -> "Monday";
            case 2 -> "Tuesday";
            case 3 -> "Wednesday";
            case 4 -> "Thursday";
            case 5 -> "Friday";
            case 6 -> "Saturday";
            case 7 -> "Sunday";
            default -> "Invalid day number";
        };
        System.out.println(name);
    }
}
```

It printed Monday for 1, Friday for 5, and Sunday for 7; for 8 and 0 it printed “Invalid day number”.

Example 7.6

A program that finds the smaller of two numbers with the conditional operator and then extends this to three numbers.

Solution. The smaller of two numbers is `a < b ? a : b`. For three numbers, first the smaller of the first two is found, then this value is compared with the third. Using two separate variables is more readable than a one-line nested conditional operator.

```
import java.util.Scanner;

public class Smallest {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int a = in.nextInt(), b = in.nextInt(), c = in.nextInt();
        int ofTwo = a < b ? a : b;
        int ofThree = ofTwo < c ? ofTwo : c;
        System.out.println(ofTwo + " " + ofThree);
    }
}
```

For (7, 12, 5), the smaller of the first two is 7 and the smallest of the three is 5; the program printed “7 5”. For (4, 4, 9) it printed “4 4”, and for (−1, 8, −6) it printed “−1 −6”.

Example 7.7

A check for “score outside the range 0–100” is to be added to the letter-grade program. Where in the chain should this check be written?

Solution. At the very beginning of the chain. If the check were put at the end, an invalid score, for example 105, would satisfy the first condition (*105 >= 90*) and receive an A; the check would never get its turn. −5 would similarly receive an F. Filtering out invalid cases at the start of the chain makes the later conditions work only with valid scores.

```
import java.util.Scanner;

public class GradeWithRange {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int score = in.nextInt();
        if (score < 0 || score > 100)
            System.out.println("Invalid score");
        else if (score >= 90)
            System.out.println('A');
        else if (score >= 80)
            System.out.println('B');
        else if (score >= 70)
            System.out.println('C');
        else
            System.out.println('F');
    }
}
```

The program printed “Invalid score” for 105 and −5, A for 100 and 95, C for 75, and F for 0. The boundary values 0 and 100 are considered valid; because `<` and `>` are used in the condition, these two values do not fall into the “invalid” branch.

7.10 Summary

- Every algorithm can be written with three flows: the **sequence structure**, the **selection structure**, and the **repetition structure**. In a sequence, statements run once each in the order written; each statement uses the values left by the previous ones.

- A **condition** is a boolean expression that produces the value `true` or `false`. Equality is tested with `==`; `=` is an assignment, and `if (x = 5)` does not compile. The notation `10 < x < 20` does not compile either; the correct form is `x > 10 && x < 20`.
- `if` runs its body if the condition is true; **if-else** runs exactly one of its two branches. Without braces, the body is a single statement; the compiler does not look at indentation. The semicolon in a line `if (…);` is an **empty statement**.
- In an **else-if chain**, the conditions are tried from top to bottom; the first true branch runs and the rest are skipped. In a chain built on lower bounds, the thresholds are written from largest to smallest. Independent ifs, on the other hand, are all checked.
- If the final `else` is deleted, a variable assigned in the branches causes the "might not have been initialized" error. Java does not allow an unassigned local variable to be read.
- In a **nested decision**, the inner condition is checked only when the outer condition selects that branch. Deep nesting can be flattened with `&&`. A **dangling else** attaches to the nearest `if` that has no else; the ambiguity is removed with braces.
- `&&` requires both sides to be true, `||` requires at least one side to be true; `!` reverses. Both **short-circuit**: if the result is already determined by the left side, the right side is not evaluated. That is why `b != 0 && a / b > 1` is safe. The single-character `&` and `|` always evaluate both sides.
- **De Morgan's laws**: `!(a && b)` = `!a || !b`, `!(a || b)` = `!a && !b`. The opposites of the comparisons: `>=` ↔ `<`, `>` ↔ `<=`, `==` ↔ `!=`.
- The **conditional operator** `condition ? v1 : v2` is an expression that produces a value; it is used for short selections between two values and groups from the right.
- For strings, `==` compares references and `equals` compares content. A string read from the keyboard gives `false` when compared with `==` to a literal carrying the same text. Always compare strings with `equals`.
- **switch** chooses among the discrete values of a single expression; case labels are constants. A **switch expression** produces a value and must cover all values; for int and String, `default` is mandatory. The value of a branch with a block is given with `yield`.
- In a classic switch, without `break` execution **falls through** into the code of the next case. In the arrow form (`->`) there is no fall-through. For ranges and complex conditions if-else is suitable; for the discrete values of a single expression, switch is suitable.
- Lessons of the examples: the even test is done with `% 2 == 0`, because in Java `-7 % 2 = -1`. The absolute value of `Integer.MIN_VALUE` **overflows** in int. `Math.sqrt` gives `NaN` for a negative number, and floating-point division by zero gives `Infinity`. In the "candidate and compare" pattern, the candidate is initialized with one of the inputs.

7.11 Key terms

The key terms of this chapter are listed below.

control flow control structure sequence structure selection structure repetition structure statement expression widening conversion trace table condition boolean comparison operator if-else statement branch block curly braces empty statement multiway decision else-if chain nested decision dangling else logical operator short-circuit evaluation De Morgan's laws operator precedence exception conditional (ternary) operator reference string literal string pool heap switch expression switch statement case label arrow form (->) fall-through remainder absolute value overflow compile error

---

7.12 Try it yourself

1. With p = 4 and q = 9, the line `if (p > 3 || ++q > 0) p = p * 2;` runs. What are p and q afterwards?

   Answer

   The left side *4 > 3* is true. `||` short-circuits and `++q` never runs; q stays 9. Because the condition is true, *p = 4 × 2 = 8*. Result: p = 8, q = 9.

2. With t = 15, what is the value of the expression `t % 3 == 0 && t % 5 == 0 ? "FizzBuzz" : t % 3 == 0 ? "Fizz" : "other"`? What happens for t = 9 and t = 7?

   Answer

   The conditional operator groups from the right; the expression reads “if the first condition is true, FizzBuzz; otherwise (if the second condition is true, Fizz; otherwise other)”. 15 is divisible by both 3 and 5: “FizzBuzz”. 9 is divisible by 3 but not by 5: the first condition is false and the second is true, “Fizz”. 7 is divisible by neither: “other”.

3. What does the following code assign for score = 72? What happens for score = 90 and score = 40?

```
if (score >= 50) {
    if (score >= 85) grade = "Excellent";
    else grade = "Pass";
} else grade = "Fail";
```

   Answer

   For 72, the outer condition *72 >= 50* is true and the inner condition *72 >= 85* is false: grade = "Pass". For 90, both conditions are true: "Excellent". For 40, the outer condition is false and the inner if is never checked: "Fail". The braces attach the inner `else` to the inner `if` and the outer `else` to the outer `if`.

4. How can the condition `if (!(temp < 15 || rain))` be written so that the `!` operator remains only in front of the boolean variable? Is the condition true for temp = 30, rain = false?

   Answer

   By De Morgan's law, `||` turns into `&&` and each term is negated: `temp >= 15 && !rain`. The opposite of `<` is `>=`. For 30 and false, *30 >= 15* is true and `!false` is true; the condition is true. In the original form too, *30 < 15* is false, rain is false, their OR is false, and its negation is true.

5. A classic switch has the lines `case 1: score += 10;`, `case 2: score += 20; break;`, `case 3: score += 30;`, `default: score += 5;`. score starts at 0 each time. What is score for k = 1, 2, 3, and 4?

   Answer

   k = 1: there is no break in case 1, so execution falls into case 2: *10 + 20 = 30*, then break. k = 2: only 20. k = 3: there is no break in case 3, so execution falls into default: *30 + 5 = 35*. k = 4: no case matches, default: 5. Result: 30, 20, 35, 5.

6. With `char grade = 'B';`, what is the value of `switch (grade) { case 'A', 'B' -> "good"; case 'C' -> "fair"; default -> "poor"; }`? What happens if the `default` line is deleted from this expression?

   Answer

   'B' is in the list of the first case: the value is "good". There is no fall-through in the arrow form; the second case does not run. If default is deleted, the program does not compile: a char can take more than 65 thousand values and a switch expression must cover all values. The compiler gives the message “the switch expression does not cover all possible input values”.

7. With a = 5 and b = 0, what do the lines `if (b == 0 || a / b > 2)` and `if (b == 0 | a / b > 2)` do?

   Answer

   In the first line the left side is true; `||` short-circuits and the division is never evaluated. The condition is true and the body runs. In the second line, `|` evaluates both sides: the integer division `5 / 0` throws `ArithmeticException: / by zero` and the program stops.

8. The user types “Ada” on the keyboard and this text is read into the variable `name`. What are the

values of `name == "Ada"`, `name.equals("ada")`, and `name.equalsIgnoreCase("ada")`?

Answer

`name == "Ada"` is false: the string read from the keyboard is a new object, while the literal is in the pool; the references differ. `name.equals("ada")` is false: the contents differ in letter case ("A" ≠ "a"). `name.equalsIgnoreCase("ada")` is true: letter case is ignored.

7.13 Further reading

- Oracle, *The Java Tutorials*, the pages "The if-then and if-then-else Statements" and "The switch Statement": Java's decision structures explained with official examples and short descriptions.
- Oracle, *Java Language Specification, Java SE 21 Edition*, Chapter 14 ("Blocks, Statements, and Patterns") and 15.28 ("switch Expressions"): the precise definitions of the dangling else rule, the exhaustiveness requirement of switch expressions, and `yield`.
- R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), Section 1.3: conditionals and loops explained with short examples from scientific computing; the quadratic equation example is worked there as well.
- A. B. Downey, C. Mayfield, *Think Java* (2nd ed., O'Reilly, 2019), the chapter "Conditionals and Logic": conditions, logical operators, and nested decisions explained plainly for beginners; the online version of the book is free.

7.14 References

1. P. Deitel, H. Deitel, *Java: How to Program, Early Objects*, 11th ed., Pearson, 2018.
2. Y. D. Liang, *Introduction to Java Programming and Data Structures*, 13th ed., Pearson, 2024.
3. Oracle, *The Java Tutorials: The if-then and if-then-else Statements; The switch Statement*.
4. Oracle, *Java Language Specification, Java SE 21 Edition*, Chapter 14: Blocks, Statements, and Patterns.
5. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 1.3 Conditionals and Loops.
6. A. B. Downey, C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019 (free online version).

Loops

The real power of a computer comes from its ability to do the same job thousands of times without getting tired. In a program, loops provide this repetition. This chapter first introduces the four parts found in every loop: initialization, condition, body, and update. Then come Java's three loops: `for`, `while`, and `do-while`. Each is traced iteration by iteration with a small program. The `break` and `continue` statements, which change the flow of a loop, and typical mistakes such as infinite loops and boundary errors are also treated in detail. The second half of the chapter is devoted to examples. Loops that work with the digits of a number, loops that search for divisors, loops that read input from the user, and patterns drawn with nested loops are gathered into a few common templates. All Java programs were compiled and run with JDK 21; the outputs given in the chapter were taken from these runs.

LEARNING OUTCOMES

Loops

Point out the initialization, condition, body, and update parts of a loop in code; choose the one among `for`, `while`, and `do-while` that suits the problem; explain how `break`, `continue`, and labeled `break` change the flow.

Patterns

Compute sums, products, and counts with counters and accumulators; solve digit-count, digit-sum, reversal, and palindrome problems with the digit pattern (`% 10`, `/ 10`); find prime and perfect numbers with the divisor pattern.

Nested loops

Produce row-and-column layouts and star or number patterns with nested loops; find the formula that ties the number of spaces and characters in each row of a pattern to the row number.

Tracing

Run a loop iteration by iteration with a trace table; find mistakes such as infinite loops, boundary errors, and the `double` counter error; test a loop with normal, zero, one, negative, and extreme values.

8.1 What is a loop?

8.1.1 Why do we need repetition?

Consider a program that prints the sum of the numbers from 1 to 4. A one-line solution comes to mind at once: `sum = 1 + 2 + 3 + 4;`. When the same job is wanted for 1 to 1000, this way does not work. Writing a line with a thousand terms is both tedious and error-prone. Moreover, the upper limit is often not known when the program is written; the user enters it while the program runs.

What these problems have in common is that the same small job is done over and over: "add one more number to the sum, then move on to the next number". In programming languages, the construct that expresses this kind of repetition is called a loop. The programmer writes the job to be repeated once and says when to stop. The computer does the rest.

Definition 8.1

A **loop** is a construct that runs a group of statements over and over for as long as a condition is true. The repeated statements are called the **loop body**, and one run of the body is called an **iteration**.

Java has three loops: `for`, `while`, and `do-while`. All three can do the same job; the differences between them are when the condition is asked and where the parts are written. That is why we first need to get to know the parts common to every loop.

8.1.2 The four parts of a loop

Every loop consists of four parts. In the slides and in this chapter, each part is always shown in the same color. Figure 8.1 gathers these parts in a general flowchart.

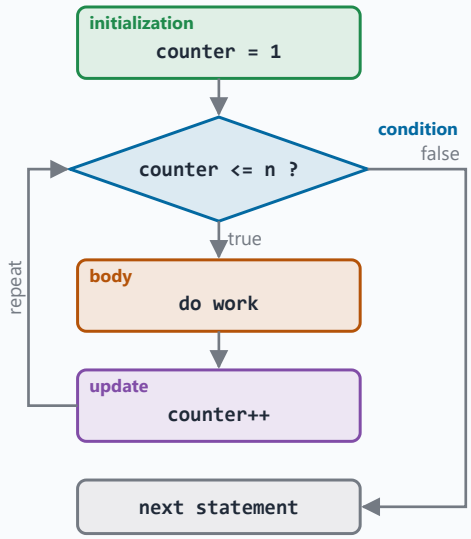


Figure 8.1: The general loop flow. The initialization runs once. As long as the condition is true, the body and the update run, and then the arrow returns to the condition. When the condition is false, execution moves to the statement after the loop.

1. **Initialization**: runs once before the loop and gives starting values to the variables the loop will use. `counter = 1` in the figure is such a step.
2. **Condition**: a logical expression whose result is `true` or `false`. In `for` and `while` loops it is asked before every iteration, and in the `do-while` loop after every iteration. If the result is `false`, the loop is left.
3. **Body**: the work done in every iteration. In the figure it is shown by the “do work” box.
4. **Update**: changes the variable used in the condition. For example, `counter++` increases the counter by 1. The job of the update is to bring the condition a little closer to its end in every iteration. If the update is forgotten, the condition always stays the same and the loop never ends.

The condition in the figure is the question `counter <= n ?`. counter starts at 1 and increases by 1 in every iteration. Eventually it passes n and the condition becomes false. This progress is what guarantees that the loop ends. When writing a loop, it should be clearly visible where each of the four parts is. If one of the parts is missing or wrong, the loop either does not run at all, runs one iteration too few or too many, or never ends.

Definition 8.2

A **counter** is a variable that changes by a fixed amount (usually 1) in every iteration and shows how many iterations have run or which element is next. An **accumulator** is a variable that, in every iteration, adds a new value to itself or multiplies itself by it. A sum accumulator starts from 0 and a product accumulator from 1.

The reason for the starting values is simple. 0 is the identity element of addition: adding a number to 0 gives that number. The identity element of multiplication is 1. If a product accumulator is mistakenly started at 0, every iteration multiplies by 0, so the result is always 0.

8.2 The for loop

8.2.1 Syntax and order of execution

When the number of repetitions is known in advance, the most suitable loop is the `for` loop. Three of the four parts, initialization, condition, and update, are written on the same line, inside the parentheses after the word `for`. The parts are separated by semicolons. The body goes between braces. The following program adds the numbers from 1 to n:

```
int n = 4;
int sum = 0;
for (int i = 1; i <= n; i++) {
    sum += i;
}
System.out.println(sum);
```

When the program runs, only the sum is printed on the screen:

```
10
```

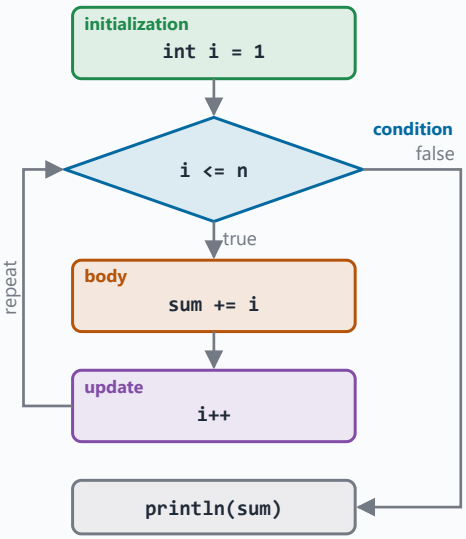


Figure 8.2: The flowchart of the for loop. The three parts in the header are spread over separate boxes in the diagram: `int i = 1` is the initialization, `i <= n` is the condition, and `i++` is the update. The body is the statement `sum += i`.

Figure 8.2 shows the flow of this program. Although the parts of a for loop are written side by side in the code, they do not run in the order in which they are written. The actual order of execution is as follows:

1. `int i = 1` runs. This step happens only once, when entering the loop.
2. The condition `i <= n` is asked. If it is true, the body is entered; if it is false, the loop ends.
3. The body `sum += i` runs.
4. The update `i++` runs and the flow returns to step 2, the condition.

In short, the order is "initialization → condition → body → update → condition → body → update …". The update is written before the body inside the parentheses but always runs after the body. This detail is very important when tracing a loop by hand.

The variable `i` defined in the header has one more property: it lives only inside the loop. The region of the program in which a variable can be used is called its **scope**. The scope of an i defined as `for (int i = 1; …)` is the loop itself. If `System.out.println(i);` is written after the loop, the compiler does not recognize i and reports an error. If the value of i is needed after the loop, i must be defined before the loop: `int i; for (i = 1; i <= n; i++) …`.

8.2.2 Tracing a for loop step by step

The most reliable way to understand what a loop does is to run it on paper like a computer. The values that the variables take in each iteration are written in a table. This table is called a **trace table**. Table 8.1 is the trace table of the program above. The table was produced by running the same program with JDK 21 and printing the variables in each iteration.

Iteration	i	i <= n	sum (after the body)
1	1	`true`	1
2	2	`true`	3
3	3	`true`	6
4	4	`true`	10
—	5	`false`	10 (the loop ends)

Table 8.1: 1 + 2 + 3 + 4 with `for` (n = 4). The last row is the condition check that ends the loop.

The table is read as follows. When entering the loop, `int i = 1` runs. In iteration 1 the condition *1 <= 4* is true and the body makes *sum = 0 + 1 = 1*. Then `i++` makes i 2 and the flow returns to the condition. In iteration 2, *2 <= 4* is true and *sum = 1 + 2 = 3*. In the same way, sum becomes 6 in iteration 3 and 10 in iteration 4. The `i++` at the end of iteration 4 makes i 5. The condition *5 <= 4* is false and the loop ends. The `println` after the loop prints 10.

Two counts come out of the table. The body ran 4 times, and the condition was asked 5 times. The last check of the condition, the one that came out false, is also an evaluation. In general, in a for loop where `i` increases from 1 to n, the body runs n times and the condition runs n + 1 times. The initialization runs only once.

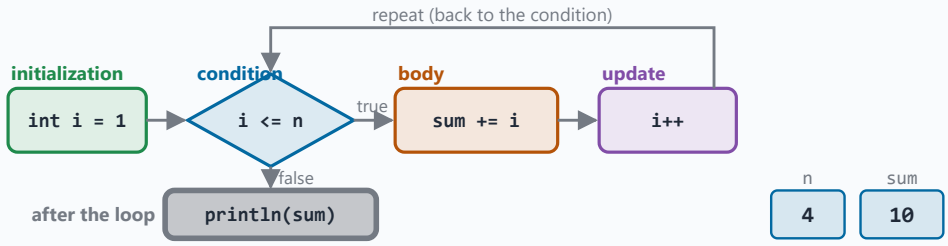


Figure 8.3: The last step of the trace. Because the condition is false, the flow has moved to the "after the loop" box; n = 4 and sum = 10. i is no longer visible, because its scope ended with the loop.

Common mistake

Defining the accumulator inside the loop. If the line `int sum = 0;` is moved into the body, two problems arise. First, every iteration resets the sum to 0 and the work of the earlier iterations is lost. Second, the scope of the sum is limited to the body and the `println(sum)` after the loop does not compile. The accumulator is always defined before the loop and gets its starting value there.

8.2.3 Flexible forms of the for header

The three parts of the for header do not always have to be of the form "start at 1, increase by 1". The counter can start from any value, increase by twos, or count backward. In the initialization part, several variables of the same type can be defined; in the update part, several expressions separated by commas can be written. Table 8.2 gives three example headers and the values they produce.

Header	Behavior
`for (int i = 0, j = 9; i < j; i++, j--)`	i and j move together: (0,9) (1,8) (2,7) (3,6) (4,5); 5 iterations
`for (int i = 0; i <= 10; i += 2)`	0 2 4 6 8 10
`for (int i = 5; i >= 1; i--)`	countdown: 5 4 3 2 1

Table 8.2: Flexible forms of the for header (the values were obtained by running on JDK 21).

In the first row there are two counters: i advances from the start and j from the end. In every iteration i increases by 1 and j decreases by 1. The condition `i < j` stops the loop when the two counters meet. After the iteration (4,5), i = 5 and j = 4; *5 < 4* is false. This pattern is often used in algorithms that move from both ends of an array toward the middle.

Each of the three parts is optional. All three can even be left empty. If the condition part is empty, Java treats it as `true` and the loop does not end on its own. In such a loop the exit must be provided by a `break` in the body:

```
int k = 0;
for (;;) {            // all parts empty: always true
    k++;
    if (k == 3) break;  // exit is required
}
System.out.println(k);   // 3
```

k increases by 1 in every iteration and when it reaches 3, `break` ends the loop. The program prints 3. That the parts can be left empty is a convenience but reduces readability. For a job with a counter, an ordinary for header with all three parts filled in is always clearer.

8.2.4 i++ versus ++i

Java has two short ways of increasing a variable by 1: `i++` (post-increment) and `++i` (pre-increment). Both increase i by 1. The difference is in the *value* of the expression. The expression `i++` gives the old value of i and then increases i. `++i` first increases i and then gives the new value. Table 8.3 shows this difference with four small examples.

Initially	Expression	Result	After
a = 5	`b = a++`	b = 5	a = 6
a = 5	`b = ++a`	b = 6	a = 6
idx = 0	`d[idx++]`	10	idx = 1
idx = 0	`d[++idx]`	20	idx = 1

Table 8.3: Post-increment and pre-increment (d = {10, 20, 30}; the values were verified with JDK 21).

In the first two rows a becomes 6 in both cases; only the value that goes to b differs. In the last two rows idx becomes 1 in both cases. `d[idx++]` uses the old index (0) and gives 10. `d[++idx]` first raises the index to 1 and gives 20.

In the update part of a for header, however, the value of the expression is not used anywhere; only its side effect, the increase of i, matters. That is why there is no difference between the following two loops:

```
for (int i = 0; i < 3; i++) System.out.print(i + " ");   // 0 1 2
for (int i = 0; i < 3; ++i) System.out.print(i + " ");   // 0 1 2
```

Both print `0 1 2`. For readability, `i++`, the customary form for counters, is preferred. The difference between the two forms appears only when the value of the increment expression is used inside another expression. Such nested use makes code harder to read and is a habit to avoid.

8.3 The while loop

8.3.1 Syntax and order of execution

The for loop offers a comfortable notation when the number of repetitions is known. In some problems, however, the number of iterations needed is not known in advance. Processing the digits of a number one by one is like that: one iteration is needed for 7 and three for 725. Asking questions until the user makes the right guess is also like that. In these cases the `while` loop is a more natural choice. After the word while, only the condition is written in parentheses. The initialization comes before the loop and the update is inside the body.

Definition 8.3

The `while` **loop** is a loop written as `while (condition) { body }` that asks the condition *before* every iteration. The body runs as long as the condition is true. Loops whose condition is asked before the body are called **pre-test loops**; the for loop is also a pre-test loop.

The following program computes the value of 5!, that is, the product *5 × 4 × 3 × 2 × 1*:

```
int num = 5;
int factorial = 1;
while (num > 1) {
    factorial *= num;
    num--;
}
System.out.println(factorial);
```

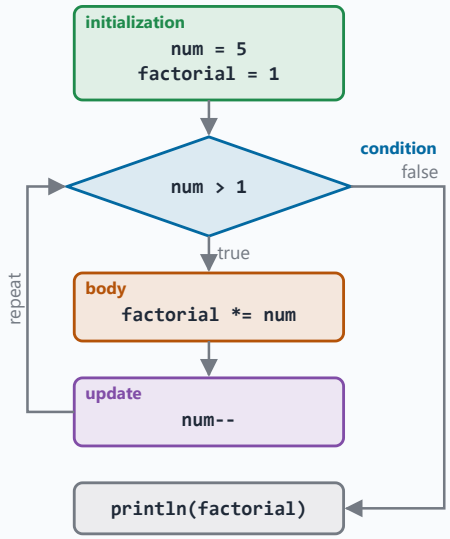


Figure 8.4: The flowchart of the while loop. The initialization (`num = 5` , `factorial = 1`) comes before the loop, and the update (`num--`) is at the end of the body.

Figure 8.4 has the same form as the for diagram. Only the way the code is written changes: the parts gathered in the header of a for loop are spread over three separate places here. This scattering is the only risk of the while loop. The update inside the body can be forgotten. If that happens, the condition never changes and the loop never ends.

Because the condition is asked before the body, the body of a while loop may never run. If `num = 1` is given in the program above, the first check *1 > 1* is false. The body is never entered and 1 is printed. This result is correct, because *1! = 1*. In the same way, 1 is printed for `num = 0` , which agrees with the definition *0! = 1*.

8.3.2 Computing 5! step by step

Iteration	num (in the condition)	num > 1	factorial	num (after the update)
1	5	`true`	5	4
2	4	`true`	20	3
3	3	`true`	60	2
4	2	`true`	120	1
—	1	`false`	120	—

Table 8.4: 5! with while. The column “num (in the condition)” is the value of num when the condition is asked.

Table 8.4 shows each iteration in one row. The product accumulator `factorial` starts at 1. In iteration 1, *5 > 1* is true; *factorial = 1 × 5 = 5* and `num--` lowers num to 4. In iteration 2, *5 × 4 = 20*, in iteration 3, *20 × 3 = 60*, and in iteration 4, *60 × 2 = 120* is computed. At the end of iteration 4, num becomes 1. The condition *1 > 1* is false and the loop ends. The program prints 120.

The body ran 4 times. The condition `num > 1` was chosen because multiplying by 1 does not change the result; `num >= 1` would have given the same result and would only have run one extra iteration. Here the counter advances downward. Because the update is `num--`, num decreases by 1 in every iteration and finally drops to 1. It is this progress that makes the loop end.

The same computation can also be written with for: `for (int i = 2; i <= 5; i++) factorial *= i;`. Both forms are correct. Because the number of repetitions is known in advance, for is slightly more readable here. The factorial grows very fast: *12! = 479,001,600* fits in an int, *13! = 6,227,020,800* does not. An int can hold at most 2,147,483,647. If 13! is computed with int, Java gives no error; the value silently wraps around and a wrong number such as 1932053504 is printed. That is why `long` is used in factorial computations. A long is enough up to 20!; 21! does not fit in a long either.

8.4 The do-while loop

8.4.1 Syntax and order of execution

In some jobs, the body has to run at least once. Consider a menu program. The user cannot make a choice before the menu is printed on the screen; and until the choice is made, the answer to the question “should we exit?” is not known. For this kind of job Java has the `do-while` loop.

> **Definition 8.4**
>
> The `do-while` **loop** is a loop written as `do { body } while (condition);` that asks the condition *after* every iteration. The body runs once without looking at the condition; the later iterations depend on the condition. Such loops are called **post-test loops**.

The following program reads a choice from the user. If the choice is 1, it prints “Hello!”. It repeats this until the user enters 0:

```
Scanner in = new Scanner(System.in);
int choice;
do {
  System.out.print("Choice (1/0): ");
  choice = in.nextInt();
  if (choice == 1)
    System.out.println("Hello!");
} while (choice != 0);
System.out.println("Goodbye");
```

The `Scanner` class is used to read from the keyboard. The line `import java.util.Scanner;` must be written at the top of the file. The call `in.nextInt()` waits until the user enters an integer and presses Enter, and returns the number entered.

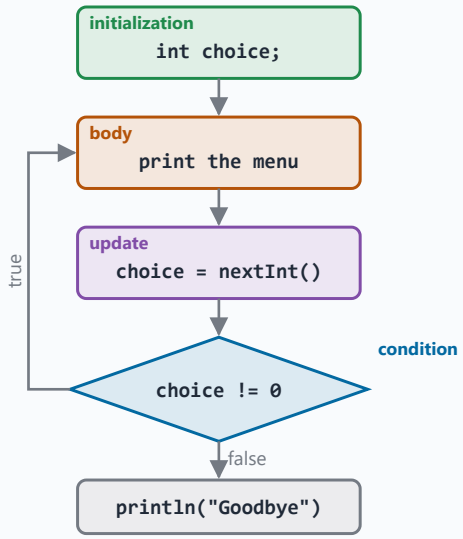


Figure 8.5: The flowchart of the do-while loop. The body (print the menu) and the update (`choice = nextInt()`) come before the condition. If the condition is true, the arrow goes back up to the start of the body.

When Figure 8.5 is compared with the for and while diagrams, the difference is seen at once: the decision box is not at the start of the loop but at its end. The variable `choice` has been declared before the loop but has not been given a value. Java does not allow an unassigned local variable to be read. No problem arises here, because the condition is asked for the first time after the body, and by that time the line `choice = in.nextInt();` has certainly run.

Two details need care in the syntax. First, there is a semicolon at the end of the line `while (choice != 0)` ; in for and while loops there is no semicolon at this place. Second, the variable used in the condition must be declared before the loop. If choice had been declared inside the body, its scope would have been limited to the body and it could not have been used in the `while` line.

8.4.2 Tracing a menu loop

Let the user enter 1, 2, and 0 in turn. Table 8.5 shows the three iterations of the loop.

Iteration	choice read	choice == 1	Printed	choice != 0
1	1	`true`	Choice (1/0): and Hello!	`true`
2	2	`false`	Choice (1/0):	`true`
3	0	`false`	Choice (1/0):	`false`

Table 8.5: The do-while menu, input 1, 2, 0. The condition is asked at the end of every iteration.

In iteration 1 the body is entered without looking at the condition. The menu is printed, the user enters 1, *choice == 1* is true, and “Hello!” is printed. At the end of the iteration *1 != 0* is true and execution returns to the start of the body. In iteration 2, 2 is read. The if condition is false and nothing

is printed; because *2 != 0* is true, the loop continues. In iteration 3, 0 is read and *0 != 0* is false. The loop ends and “Goodbye” is printed.

The body ran 3 times and the condition was also asked 3 times. In a pre-test loop, the condition is asked one more time than the body runs. In a post-test loop the two counts are equal. When the program was run on JDK 21 with the same inputs, it gave the output below. Because the inputs were redirected from a file, the numbers the user types are not visible on the screen; that is why the prompt lines fall side by side:

```
Choice (1/0): Hello!
Choice (1/0): Choice (1/0): Goodbye
```

If the user enters 0 at the first choice, the output consists of only one prompt line and “Goodbye”. The menu has still been printed once. This is the natural area of use of the do-while loop: menus and input validation. For example, when asking the user for a grade between 1 and 5, the question must be asked at least once. If the value entered is outside the range, the question is repeated: `do { grade = sc.nextInt(); } while (grade < 1 || grade > 5);`. For the input 7, 0, 3 the loop runs three iterations and exits with grade = 3.

Common mistake

Using do-while where the body must not run at all. If the condition is false on the first try, while and do-while behave differently. In the following two loops the condition is false from the start (n = m = 5, limit 3). The do-while still runs once, prints 5, and makes n 6. The body of the while loop never runs and m stays 5. The output is `5 | 6 5`. In every case where the body may run zero times, a pre-test loop is chosen.

```
int n = 5;
do {
    System.out.print(n + " ");
    n++;
} while (n < 3);

int m = 5;
while (m < 3) {
    System.out.print(m + " ");
    m++;
}
System.out.println("| " + n + " " + m);
```

8.4.3 Which loop to choose?

The three loops can be converted into one another. Every for loop can be written as a while loop, and every while loop as a for loop. A do-while can also be written as a while loop with a copy of the body placed in front of it. So the choice is not a matter of capability but of readability. The right loop answers the reader's question “when and how many times does this loop run?” by the shortest route.

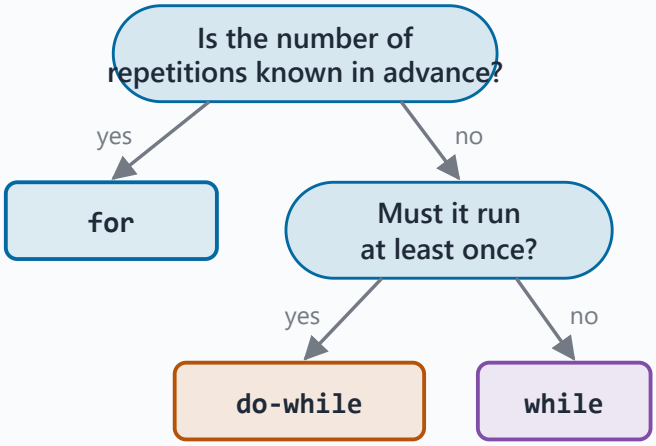


Figure 8.6: A decision tree for choosing a loop. If the number of repetitions is known in advance, for is chosen. If it is not known, we look at whether the body has to run at least once: if so, do-while; if not, while.

Situation	Choice
Count from 1 to n, repeat n times	`for`
Process digits until the number is 0	`while`
Show a menu, ask until valid input arrives	`do-while`
Add up until the user types “0”	`while` / `do-while`

Table 8.6: Typical situations and the suitable loop.

There are two options in the last row, because both solutions look natural. The solution written with while reads the first number before the loop and the next one at the end of the loop. The solution written with do-while puts the reading at the start of the body but needs a separate check in the body so that the 0 is not added to the sum. Whichever looks clearer is chosen.

8.5 break and continue

8.5.1 Skipping an iteration and leaving a loop

A loop normally ends only when its condition is false, and the whole body runs in every iteration. Two statements change this arrangement.

Definition 8.5

`break` ends the nearest enclosing loop at once; execution continues from the first statement after the loop. `continue` skips only the rest of that iteration. In a for loop the flow goes to the update (`i++`), and in a while or do-while loop to the condition; the loop continues.

The following loop counts from 1 to 10. It skips the multiples of 3 and stops when it reaches 8:

```
for (int i = 1; i <= 10; i++) {
    if (i % 3 == 0) continue;
    if (i == 8) break;
    System.out.print(i + " ");
}
System.out.println("| done");
```

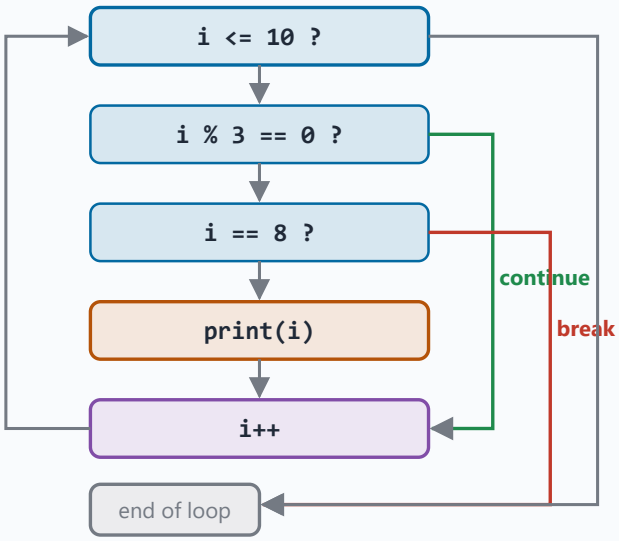


Figure 8.7: The arrows of break and continue. The continue arrow skips the `print(i)` box and goes straight to `i++`. The break arrow leaves the loop, to “end of loop”.

8.5.2 Tracing break and continue

i	i % 3 == 0	i == 8	Result	Output
1	`false`	`false`	1 is printed	1
2	`false`	`false`	2 is printed	1 2
3	`true`	not asked	continue: printing is skipped	1 2
4	`false`	`false`	4 is printed	1 2 4
5	`false`	`false`	5 is printed	1 2 4 5
6	`true`	not asked	continue: printing is skipped	1 2 4 5

i	i % 3 == 0	i == 8	Result	Output
7	`false`	`false`	7 is printed	1 2 4 5 7
8	`false`	`true`	break: the loop ends	1 2 4 5 7

Table 8.7: Trace of the loop with break and continue. The "Output" column is what has been printed so far.

Table 8.7 gives the result of the two checks for each i. For i = 1, 2, 4, 5, 7 both conditions are false and the number is printed. For i = 3 and i = 6, *i % 3 == 0* is true. continue runs, print is skipped, and the flow passes straight to `i++`. In these iterations the second if is never asked. At i = 8 the first condition is false (*8 % 3 = 2*), the second is true, and break ends the loop. 9 and 10 are never tried. Execution continues from the line after the loop and the output is:

```
1 2 4 5 7 | done
```



Figure 8.8: The fate of the numbers from 1 to 10. The green boxes are the numbers printed, and the dashed boxes are the 3 and 6 skipped by continue. The red one is 8, where break ran; 9 and 10 were never reached.

The order of the two statements matters too. In the same structure, if the conditions were `i % 4 == 0` and `i == 10` and the loop ran up to 12, 4 and 8 would be skipped and break would run at 10. The output would be `1 2 3 5 6 7 9 | done`. In this loop, for i = 8 the continue check is made first, so break never gets its turn in that iteration.

break and continue can shorten loops but also make the flow harder to trace. In a loop that exits from the middle of the body, looking at the header is not enough to see when the loop ends; the whole body has to be read. That is why these statements are used sparingly. If the same job can be written clearly with a condition or a flag variable, that way is preferred. The condition `prime && i <= n / i` in the prime test is an example of this.

8.5.3 Labeled break

A plain break ends only the nearest enclosing loop. In two nested loops, a break in the inner loop does not stop the outer loop. When it is necessary to leave both loops at once, Java's **labeled break** statement is used. A name and a colon are written in front of the outer loop; `break name;` ends the loop that carries 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 + " " + j);
    }
}
System.out.println("done");
```

The program prints the pairs of i and j from 1 to 3 and stops at the first pair whose product is 4. Table 8.8 shows side by side how the same loop behaves with the labeled and the plain break.

i	j	i * j	with break outer	with plain break
1	1	1	“1 1” is printed	“1 1” is printed
1	2	2	“1 2” is printed	“1 2” is printed
1	3	3	“1 3” is printed	“1 3” is printed
2	1	2	“2 1” is printed	“2 1” is printed
2	2	4	exits both loops	the inner loop ends
2	3	6	not tried	not tried
3	1	3	not tried	“3 1” is printed
3	2	6	not tried	“3 2” is printed
3	3	9	not tried	“3 3” is printed

Table 8.8: The result of the labeled and the plain break for each pair (i, j).

For i = 1, the product is not 4 for j = 1, 2, 3 and three lines are printed. For i = 2, j = 1, the product is 2 and “2 1” is printed. At i = 2, j = 2, $2 \cdot 2 = 4$. The labeled `break outer` leaves both loops at once; i = 3 is never tried. The flow jumps to the line after the outer loop and “done” is printed. The output is 4 number lines and “done”:

```
1 1
1 2
1 3
2 1
done
```

If the label is removed and a plain `break` is written, at i = 2, j = 2 only the inner loop ends. The outer loop moves to 3 with i++ and the inner loop starts again from j = 1. For 3 the products are 3, 6, 9; none of them is 4 and three more lines are printed. With the plain break, 7 number lines come out:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
done
```

The labeled break is useful when something is searched for in nested loops and the whole search must end the moment it is found. The same job can also be done with a flag: when the item is found, `found = true` is set and `!found &&` is added to the condition of the outer loop. The label can be any valid Java name; common choices are descriptive names such as `outer` or `search`.

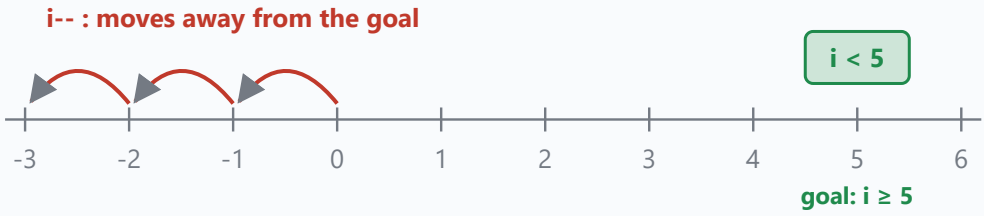
8.6 Loop errors and verification

Loop errors are often not caught by the compiler. The program compiles, runs, and either gives a wrong result or never ends. This section treats four typical problems: infinite loops, `double` counters, a broken loop invariant, and boundary errors.

8.6.1 Infinite loops

Definition 8.6

An **infinite loop** is a loop that does not end on its own because its condition never becomes false. The program seems “frozen”: it either prints nothing or prints the same lines endlessly.




```
for (int i = 0; i < 5; i--) → i = 0, -1, -2, … condition always true

while (i < 5) { … } // no i++ → i is always 0
```

Figure 8.9: The counter goes in the opposite direction from the goal. With the condition `i < 5`, the counter decreases with `i--` as 0, −1, −2, … and moves away from the goal. In the while loop below, there is no update at all and i stays 0.

Figure 8.9 shows two typical causes. In the first, the update goes in the wrong direction. In the loop `for (int i = 0; i < 5; i--)`, i decreases in every iteration and *i < 5* stays true. In the second, the update has not been written at all. If the body of `while (i < 5) { … }` has no statement that changes i, the condition keeps its value from the first iteration.

The first loop is infinite in theory. In reality, however, int is a bounded type. When i, after dropping to −2,147,483,648, is decreased once more, **overflow** occurs and the value wraps around to the largest int, 2,147,483,647. This value is not less than 5 and the loop ends. The number of iterations counted with JDK 21 is below:

Loops	Iterations	Last i processed
for (int i = 0; i < 5; i--)	2147483649	−2147483648

Table 8.9: Number of iterations of a loop whose update goes in the wrong direction (counted with a long counter).

i takes every value from 0 down to −2,147,483,648 once. That is $2^{31} + 1 = 2{,}147{,}483{,}649$ iterations, about 2.1 billion. A modern computer can finish this many iterations in a few seconds; if there is printing in the body it takes hours. In practice this loop too counts as infinite and is not a correct program.

Sometimes a loop is deliberately written as infinite: `while (true)`. Servers, game loops, and reading with a sentinel value use this form. The body of such a loop must certainly have a `break` condition. If a program gets stuck in an infinite loop, it is stopped on the command line with the Ctrl+C keys; development environments have a stop button for this.

Common mistake

Forgetting the braces. In the line `int i = 0; while (i < 5) System.out.print(i); i++;`, the body is only the statement `print(i)`. Indentation, or being written on the same line, does not make `i++` part of the body. i stays 0 and 0 is printed endlessly. Using braces in loops even when the body is a single statement prevents this mistake from the start.

8.6.2 The double counter trap

The following loop starts x at 0 and increases it by 0.1. The aim is to run 10 iterations: x = 0.0; 0.1; …; 0.9.

```
int n = 0;
for (double x = 0; x < 1.0; x += 0.1) {
    n++;
}
System.out.println(n);
```

The program prints 11, not 10:

```
11
```

The cause is that the number 0.1 cannot be represented exactly in binary. In the decimal system 1/3 = 0.333… goes on forever. In the same way, in binary 1/10 = $0.0001100110011\ldots_2$ goes on forever. double cuts this expansion off at 53 bits. A small rounding error arises in every addition and these errors accumulate. Table 8.10 gives the actual value of x in each iteration in the form Java prints it.

Iteration	x (when the condition is asked)	x < 1.0	n (after the body)
1	0.0	`true`	1
2	0.1	`true`	2
3	0.2	`true`	3
4	0.30000000000000004	`true`	4
5	0.4	`true`	5
6	0.5	`true`	6
7	0.6	`true`	7
8	0.7	`true`	8
9	0.7999999999999999	`true`	9
10	0.8999999999999999	`true`	10
11	0.9999999999999999	`true`	11
—	1.0999999999999999	`false`	11 (loop ends)

Table 8.10: A double counter advancing with `x += 0.1`. The values are the `println` output of JDK 21.

The first deviation is seen at iteration 4: x = 0.30000000000000004. At iteration 9, x = 0.7999999999999999. After 10 additions, x is not exactly 1.0 but 0.9999999999999999. Because this value is less than 1.0, the condition comes out true and the loop runs one extra iteration. When x becomes 1.0999999999999999, the loop ends.

If the condition `x != 1.0` had been written, the situation would be worse. Because x is never exactly 1.0 in any iteration, the condition stays true forever and the loop never ends. In a trial, after 1000 iterations x is about 99.9999999999986. The lesson is clear: the loop counter should be an integer. If a real number is needed, it is computed from the counter in each iteration: `for (int k = 0; k < 10; k++) { double x = k * 0.1; … }`. This loop runs exactly 10 iterations.

Not every decimal step is faulty. 0.25 = 1/4 is represented exactly in binary (0.01_2). The loop `for (double d = 0; d != 1.0; d += 0.25)` runs exactly 4 iterations and ends. But thinking each time about which numbers can be represented exactly is error-prone. The safe rule is still the int counter.

8.6.3 The loop invariant

It is useful to try a loop with a few inputs, but it is impossible to try every input. One way to see that a loop gives the right result for every input is to find a property that stays true unchanged in every iteration.

Definition 8.7

A **loop invariant** is a property that is true at the start of every iteration of the loop. It is true when entering the loop; an iteration does not break it; and when the loop ends it is still true and at that moment gives the desired result.

In the sum loop from 1 to n, the invariant is this: at the start of every iteration, *sum = 1 + 2 + ... + (i − 1)*. That is, sum is the sum of the numbers processed so far. Table 8.11 checks this property at the start of every iteration for n = 4.

i at the start of the iteration	sum	1 + ... + (i − 1)	Invariant
1	0	0 (empty sum)	true
2	1	1 = 1	true
3	3	1 + 2 = 3	true
4	6	1 + 2 + 3 = 6	true
5	10	1 + 2 + 3 + 4 = 10	true

Table 8.11: Checking the invariant in the sum loop (n = 4). The last row is the moment the condition comes out false.



Figure 8.10: Two moments of the invariant. On the left, the start of the iteration with i = 3: sum = 3 = 1 + 2. On the right, i = 5, the moment the condition comes out false: sum = 10 = 1 + 2 + 3 + 4.

The correctness of the invariant is shown in three steps:

1. **Initialization.** Before the first iteration, i = 1 and sum = 0. *1 + ... + 0* is an empty sum with no terms, and its value is 0. The invariant is true from the start.
2. **Maintenance.** Suppose that at the start of an iteration *sum = 1 + ... + (i − 1)*. The body adds i to the sum: *sum = 1 + ... + (i − 1) + i*. The update increases i by 1. Written with the new i, this is again of the form *1 + ... + (i − 1)*. The invariant is true at the start of the next iteration too.
3. **Termination.** The loop ends when *i = n + 1*. Writing n + 1 in place of i in the invariant gives *sum = 1 + 2 + ... + n*. This is exactly the desired result.

None of the three steps depends on a specific value of n. That is why the argument holds for every n. In the exponentiation loop the invariant is “at the start of every iteration *result = base*^{i}”, and in the factorial loop it is “factorial is the product of the numbers multiplied so far”. Being able to state the invariant in one sentence when writing a loop means really knowing what the loop does.

8.6.4 Boundary errors

The most common loop error is running one iteration too few or too many. Should the counter start at 0 or 1, should the condition be `<` or `<=`? A mistake in these small choices causes the loop to skip the first or last element or to process an extra one.

Definition 8.8

An **off-by-one error** is an error that appears when a loop runs one iteration fewer or one more than the correct number. It mostly arises from choosing the starting value (0 or 1) or the comparison operator (`<` or `<=`) wrongly.

Let the task be counting from 1 to 5. Figure 8.11 and Table 8.12 show which values four headers process.

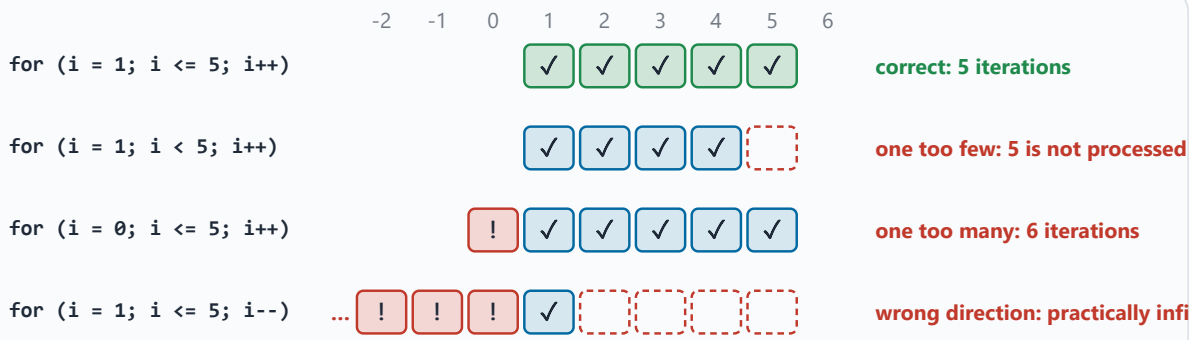


Figure 8.11: The numbers processed by four for headers. Only the first processes exactly 1–5. The second misses 5, the third processes an extra 0, and the fourth goes in the wrong direction.

Header	i values processed	Iterations
for (i = 1; i <= 5; i++)	1 2 3 4 5	5
for (i = 1; i < 5; i++)	1 2 3 4	4
for (i = 0; i <= 5; i++)	0 1 2 3 4 5	6
for (i = 1; i <= 5; i--)	1 0 −1 −2 … −2147483648	2147483650

Table 8.12: Four headers and their iteration counts (counted with JDK 21).

The last header is the same overflow as in the section on infinite loops. i starts at 1 and goes down to −2,147,483,648; that makes 2,147,483,650 iterations. Then the value wraps around and the loop ends.

Another frequently seen pattern is counting from 0. Arrays and strings are numbered from 0. That is why the form `for (int i = 0; i < 5; i++)` is very common in Java programs. This loop runs exactly 5 iterations for i = 0, 1, 2, 3, 4:

```
int count = 0;
for (int i = 0; i < 5; i++) {
    System.out.print(i + " ");
    count++;
}
System.out.println("| rounds: " + count);
```

```
0 1 2 3 4 | rounds: 5
```

If the condition `i <= 5` is written, i = 5 is processed too and the loop runs 6 iterations. The counter becomes 6 instead of 5:

```
0 1 2 3 4 5 | rounds: 6
```

The way to prevent boundary errors is a simple check. After a loop is written, three questions are asked: what value does the counter take in the first iteration, what value does it take in the last, and how many iterations does it run in total? In the form `for (int i = a; i < b; i++)` the number of iterations is $b - a$, and in the form `for (int i = a; i <= b; i++)` it is $b - a + 1$. These two formulas catch most boundary errors without running the code.

8.7 The digit pattern

Most problems solved with loops rest on a few common patterns. Once a pattern is understood, the same skeleton is reused in different problems with small changes. All the examples in this section process the digits of an integer one by one. Counting digits, summing digits, reversal, palindromes, Armstrong numbers, and decimal-to-binary conversion are variations of the same loop.

Definition 8.9

A **pattern** is a solution skeleton that recurs in many problems. A loop pattern determines the type of the loop, its starting values, its condition, and its update; from problem to problem only the operation in the body changes.

8.7.1 % 10 and / 10

Two integer operations lie at the base of the pattern. For a positive integer, `num % 10` gives the remainder of dividing the number by 10, that is, the **last digit**: *725 % 10 = 5*. `num / 10` is integer division; the fractional part is discarded and the **last digit is removed**: *725 / 10 = 72*. When the two operations are applied one after the other, the number is "consumed" digit by digit from right to left:

725 → (5) → 72 → (2) → 7 → (7) → 0

When the number is 0, no digits are left to process. The skeleton of the pattern rests on this observation:

```
while (rest > 0) {
    int digit = rest % 10;   // take the last digit
    // ... do something with the digit ...
    rest /= 10;                // drop the last digit
}
```

The loop runs as many iterations as the number has digits. For a six-digit number, 6 iterations are enough. That is why the digit pattern is fast even for very large numbers: the largest int has ten digits and the loop runs at most 10 iterations.

Two details are repeated in every example. First, the loop consumes not the number itself but a copy of it (`rest`). When the loop ends, rest is 0 but the original number is preserved; in the palindrome and Armstrong tests the result is compared with it. Second, the sign of negative numbers is dropped first. In Java *−725 % 10 = −5*; the remainder takes the sign of the dividend. If the sign is not dropped, the condition `while (rest > 0)` is false from the start and the loop never runs.

8.7.2 Counting digits

```
int num = -507;
long rest = num;
if (rest < 0) rest = -rest;
int digits = 0;
do {
    digits++;
    rest /= 10;
} while (rest > 0);
System.out.println(digits);
```

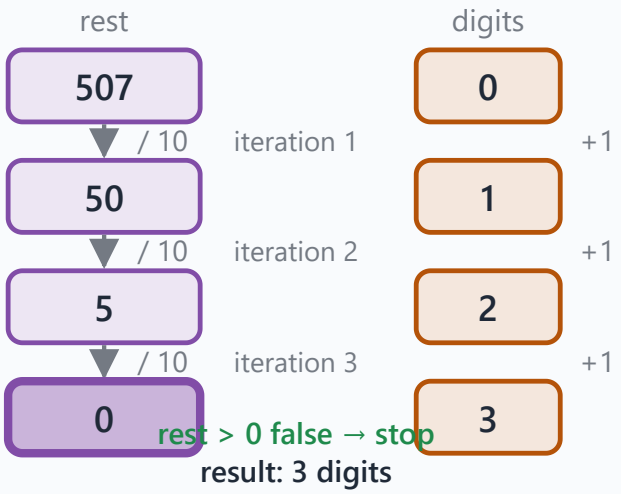


Figure 8.12: Digit count. While rest goes 507, 50, 5, 0, digits goes 0, 1, 2, 3. When rest becomes 0, the condition comes out false; the result is 3 digits.

Iteration	rest (at the start of the iteration)	digits	rest / 10	rest > 0
1	507	1	50	`true`
2	50	2	5	`true`
3	5	3	0	`false`

Table 8.13: Digit count, num = −507. After the sign is dropped, rest = 507.

Here the digit itself is not needed; only the number of divisions is counted. That is why there is no `% 10` in the body. digits is a counter and increases by 1 in every iteration. The program prints 3.

The loop being a do-while is a deliberate choice. For num = 0, a while loop would never run and would find 0 digits. Yet 0 is a one-digit number. Because do-while runs the body at least once, it finds 1 for 0 as well; no special case needs to be written. The same result can also be obtained with a while loop and a check `if (num == 0) return 1;` placed before it.

There is a reason for the copy being a `long` too. The smallest value of int is −2,147,483,648 but its largest value is 2,147,483,647. That is why the positive counterpart of −2,147,483,648 does not fit in an int. Even `Math.abs(-2147483648)` returns −2,147,483,648 again. When the copy is a long, the sign change is done safely and the correct result (10) comes out for the extreme value. If the copy were an int, the method would find 1 for this input.

8.7.3 Sum of digits

In the sum of digits, the last digit is added to an accumulator in every iteration:

```
int num = 725;
long rest = num;
if (rest < 0) rest = -rest;
int sum = 0;
while (rest > 0) {
    sum += (int) (rest % 10);
    rest /= 10;
}
System.out.println(sum);
```

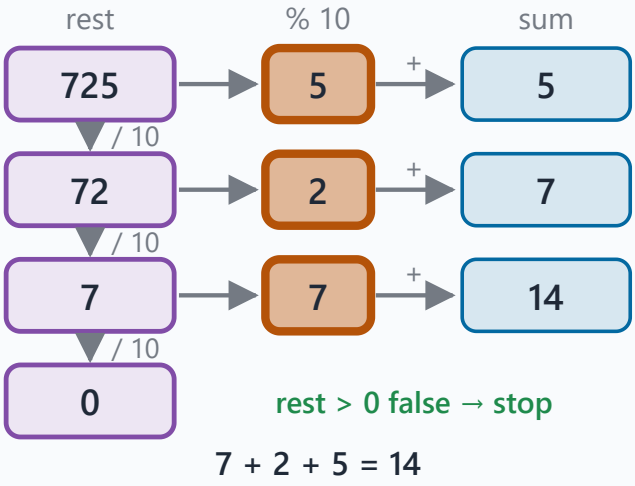


Figure 8.13: Sum of digits. From 725, `% 10` takes 5, 2, 7 in turn; the sum becomes 5, 7, 14.

Iteration	rest	rest % 10	sum	rest / 10
1	725	5	5	72
2	72	2	7	7
3	7	7	14	0

Table 8.14: Sum of digits, num = 725.

The program prints 14: *7 + 2 + 5 = 14*. A while is enough here. For num = 0 the body never runs and sum stays 0; this result is correct. The `(int)` cast is also necessary. Because rest is a long, `rest % 10` is also a long and must be explicitly converted when added to an int variable. The compound assignment `+=` would have done this conversion itself; the explicit form shows the reader what is happening. The same loop gives 14 for −725 and 2 for 1001.

8.7.4 Reversal and palindromes

To reverse a number, the digits are taken from right to left and a new number is built from left to right. In each iteration the reversed number so far is multiplied by 10; this shifts the old digits one place to the left and frees the units place. The new digit is added in the freed place.

```
int num = 1221;
int rest = num, rev = 0;
while (rest > 0) {
    int digit = rest % 10;
    rev = rev * 10 + digit;
    rest /= 10;
}
System.out.println(rev);
System.out.println(rev == num);  // palindrome?
```

Iteration	rest	digit	rev = rev × 10 + digit	rest / 10
1	1221	1	0 × 10 + 1 = 1	122
2	122	2	1 × 10 + 2 = 12	12
3	12	2	12 × 10 + 2 = 122	1
4	1	1	122 × 10 + 1 = 1221	0
rev == num	`true`			

Table 8.15: The reverse of 1221. The last row is the palindrome check.

Definition 8.10

A number whose digits stay the same when read backward is called a **palindrome**. 1221, 7, and 12321 are palindromes; 123 is not.

The program prints 1221 first and then `true`. The palindrome test checks whether the reversed number is equal to the original. That is why the loop consumes a copy, not the original number. For 123 the table looks like this:

Iteration	rest	digit	rev = rev × 10 + digit	rest / 10
1	123	3	0 × 10 + 3 = 3	12
2	12	2	3 × 10 + 2 = 32	1
3	1	1	32 × 10 + 1 = 321	0
rev == num	`false`			

Table 8.16: The reverse of 123 is 321; since 321 ≠ 123, 123 is not a palindrome.

Common mistake

Forgetting trailing zeros. The reverse of 120 comes out as 21 with this method, because there is no int such as 021. 1200 also turns into 21. The palindrome test still gives the right result for these numbers (120 ≠ 21). But if the reverse has to be shown on the screen as “021”, a

character string rather than a number must be built.

8.7.5 Armstrong numbers

Definition 8.11

A number equal to the sum of its digits each raised to the power of the number of digits is called an **Armstrong number**. For a three-digit number the power is 3: $153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27$.

```
int num = 153;
int rest = num, sum = 0;
while (rest > 0) {
    int r = rest % 10;
    sum += r * r * r;   // 3 digits: cube
    rest /= 10;
}
System.out.println(sum == num);
```

Iteration	rest	r	r × r × r	sum
1	153	3	27	27
2	15	5	125	152
3	1	1	1	153

Table 8.17: The Armstrong check of 153.

The pattern is the same as the sum of digits; only the cube of the digit is added instead of the digit. The sum reaches 153 and the program prints `true`. This code is correct only for three-digit numbers, because the power is fixed at 3. A general solution first finds the number of digits and then raises each digit to that power. When all numbers from 100 to 999 are tried, four Armstrong numbers come out: 153, 370, 371, and 407.

8.7.6 Converting decimal to binary

The most interesting variation of the pattern is changing the base. When 2 is used instead of 10, `n % 2` gives the last bit of the number in binary and `n / 2` gives the number with that bit removed. The bits are obtained from right to left. The following program builds the binary representation in an int using decimal digits (1 and 0). Each new bit is written one place to the left of the previous one; for this the variable `place` grows as 1, 10, 100, ...

```
int n = 13;
int bin = 0, place = 1;
while (n > 0) {
    bin += (n % 2) * place;
    place *= 10;
    n /= 2;
}
System.out.println(bin);   // 1101
```

Iteration	n	n % 2	digits	bin	n / 2
1	13	1	1	1	6
2	6	0	10	1	3
3	3	1	100	101	1
4	1	1	1000	1101	0

Table 8.18: The binary representation of 13.

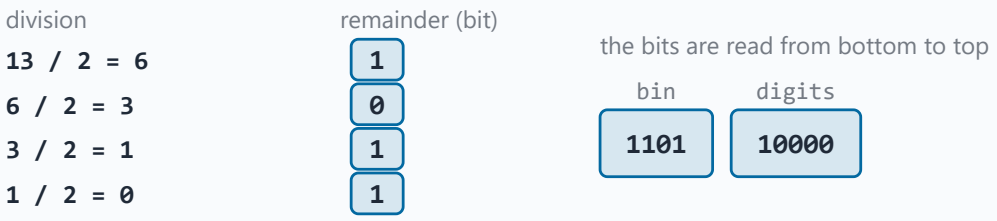


Figure 8.14: The division ladder. 13 is divided by 2 repeatedly; the remainders are 1, 0, 1, 1. The bits are read from bottom to top: 1101.

The program prints 1101. Check: $1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 13$. This method has a limit. Because the binary representation is built with decimal digits, the result grows quickly. The binary representation of 1023 is ten 1s (1111111111) and fits in an int. The binary representation of 1024 is 10000000000, that is, eleven digits, and does not fit in an int. For this input the program silently overflows and prints 1410065408. For large numbers the result is built as a character string, or Java's `Integer.toBinaryString` method is used.

8.8 Product and divisor patterns

8.8.1 Exponentiation

An integer power of a base is repeated multiplication of the base by itself: $2^4 = 2 \times 2 \times 2 \times 2$. The number of repetitions (the exponent) is known; that is why the for loop is suitable.

```
int base = 2, exp = 4;
int result = 1;
for (int i = 0; i < exp; i++) {
    result *= base;
}
System.out.println(result);
```

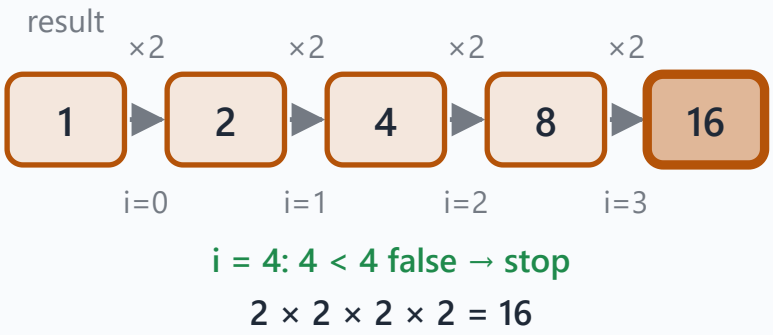


Figure 8.15: Exponentiation. result starts at 1 and doubles in 4 iterations: 2, 4, 8, 16. At i = 4 the condition comes out false.

Iteration	i	i < exp	result
1	0	`true`	2
2	1	`true`	4
3	2	`true`	8
4	3	`true`	16
—	4	`false`	16 (loop ends)

Table 8.19: The computation of 2^4.

The loop runs exactly `exp` times and multiplies by the base once in each iteration. The invariant has the form "at the start of every iteration *result* = 2^i". At i = 4 the loop ends and result = 2^4 = 16. The product accumulator starts at 1. If exp = 0, the loop never runs and the result stays 1; this agrees with $2^0 = 1$.

The code does not cover negative exponents; if exp is negative, the loop never runs and 1 is wrongly found. Large results also exceed the int limit. 2^{30} = 1,073,741,824 fits. 2^{31} comes out as −2,147,483,648 in int, and 2^{32} as 0. Overflow produces no error message; the result is silently wrong.

8.8.2 Perfect numbers

In the divisor pattern, the loop tries the possible divisors of the number one by one. If i divides n, *n % i == 0*.

Definition 8.12

A number equal to the sum of its positive divisors other than itself is called a **perfect number**. The smallest example is *6 = 1 + 2 + 3*.

```
int n = 28;
int sum = 0;
for (int i = 1; i <= n / 2; i++) {
    if (n % i == 0) sum += i;
}
System.out.println(sum == n);
```

The largest divisor of n other than itself is at most n / 2. A number greater than n / 2 and less than n cannot divide n. That is why the loop stops at n / 2 and avoids unnecessary iterations. Table 8.20 shows the 14 iterations for 28.

i	28 % i	Divisor?	sum
1	0	yes	1
2	0	yes	3
3	1	no	3
4	0	yes	7
5	3	no	7
6	4	no	7
7	0	yes	14
8	4	no	14
9	1	no	14
10	8	no	14
11	6	no	14
12	4	no	14
13	2	no	14
14	0	yes	28

Table 8.20: The divisors of 28. sum increases only in iterations where a divisor is found.

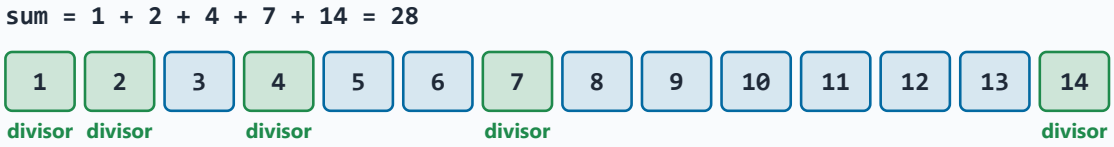


Figure 8.16: The divisor candidates of 28. The green boxes are the divisors: 1, 2, 4, 7, 14. Their sum is 28.

The divisors are 1, 2, 4, 7, and 14. Because their sum is *1 + 2 + 4 + 7 + 14 = 28*, the program prints `true`. When tried up to 10,000, only four perfect numbers come out: 6, 28, 496, and 8128.

8.8.3 The prime test

> **Definition 8.13**
>
> An integer greater than 1 that has no positive divisor other than 1 and itself is called a **prime number**. 0 and 1 are not prime; 2 is the only even prime.

```
int n = 29;
boolean prime = n >= 2;
for (int i = 2; prime && i <= n / i; i++) {
    if (n % i == 0) prime = false;
}
System.out.println(prime);
```

There is an idea in every line of the code. The assignment `prime = n >= 2` eliminates 0, 1, and negative numbers from the start. The loop searches for a divisor starting from 2. When a divisor is found, `prime = false` and the `prime &&` in the condition stops the loop. A flag is used here instead of break. The flag keeps the state in a logical variable and makes the exit condition of the loop visible in the header.

The second idea is the limit of the search. If n has a divisor other than 1 and itself, at least one of these divisors is not greater than $\sqrt{n}$. Because if $n = a \times b$, a and b cannot both be greater than $\sqrt{n}$; if they were, their product would exceed n. So it is enough to try i up to $\sqrt{n}$. The `i <= n / i` in the condition means the same as $i \times i \le n$, but `i * i` can overflow for a very large i; the division does not overflow.

n	i	n / i	prime && i <= n / i	n % i	prime
29	2	14	`true`	1	`true`
29	3	9	`true`	2	`true`
29	4	7	`true`	1	`true`
29	5	5	`true`	4	`true`

n	i	n / i	prime && i <= n / i	n % i	prime
29	6	4	`false`	—	`true`
21	2	10	`true`	1	`true`
21	3	7	`true`	0	`false`
21	4	not evaluated	`false`	—	`false`

Table 8.21: The prime test for 29 and 21. For each n, the last row is the condition check that ends the loop.

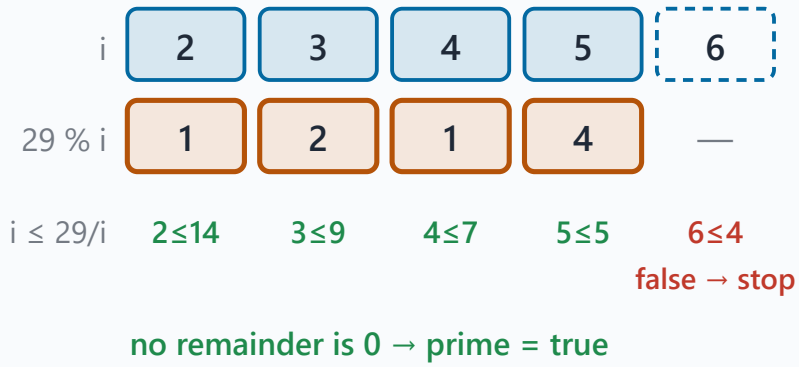


Figure 8.17: For 29, i = 2, 3, 4, 5 are tried; the remainders are 1, 2, 1, 4. At i = 6, *6 ≤ 29 / 6 = 4* is false, the loop ends, and 29 is prime.

For 29, 4 divisors are tried and none of them divides 29. At i = 6, *6 <= 29 / 6*, that is, *6 <= 4*, is false. $\sqrt{29} \approx 5.39$ has been passed and there is no need to look for a larger divisor. The program prints `true`.

For 21, at i = 3, *21 % 3 = 0* and prime = false is set. The update raises i to 4. The first part of the condition, `prime`, is now false. The `&&` operator short-circuits: if the first operand is false, the second is never evaluated. The loop ends without `i <= n / i` being tried, and the program prints `false`. Only 2 divisors have been tried.

The gain from the $\sqrt{n}$ limit is large. For 97, only 8 candidates (2…9) are tried. The largest int, 2,147,483,647, is prime; 46,339 trials are enough for this number. A loop searching from 2 to n − 1 would run more than two billion iterations for the same job.

8.8.4 Primes from 1 to n

When the prime test is put inside an outer loop, all primes from 2 to n are listed. For each k, the inner loop performs the prime test:

```
int n = 20;
for (int k = 2; k <= n; k++) {
    boolean prime = true;
    for (int i = 2; i <= k / i; i++) {
        if (k % i == 0) { prime = false; break; }
    }
    if (prime) System.out.print(k + " ");
}
```

Here the inner loop ends with break. The variable `prime` is set to `true` again for each k; this assignment must be inside the outer loop. If it is moved outside the loop, the flag stays false forever after the first composite number and nothing is printed after 2 and 3. Table 8.22 gives the divisors tried for each k.

k	Values of i tried	Result
2	none (2 > 2 / 2)	prime, printed
3	none (2 > 3 / 2)	prime, printed
4	2 (4 % 2 = 0, break)	not prime
5	2	prime, printed
6	2 (6 % 2 = 0, break)	not prime
7	2	prime, printed
8	2 (8 % 2 = 0, break)	not prime
9	2, 3 (9 % 3 = 0, break)	not prime
10	2 (10 % 2 = 0, break)	not prime
11	2, 3	prime, printed
12	2 (12 % 2 = 0, break)	not prime
13	2, 3	prime, printed
14	2 (14 % 2 = 0, break)	not prime
15	2, 3 (15 % 3 = 0, break)	not prime
16	2 (16 % 2 = 0, break)	not prime
17	2, 3, 4	prime, printed
18	2 (18 % 2 = 0, break)	not prime
19	2, 3, 4	prime, printed
20	2 (20 % 2 = 0, break)	not prime

Table 8.22: The prime test from 2 to 20 (the values of i tried in the inner loop).

For 2 and 3 the inner loop never runs, because *2 <= k / 2* is false (2 / 2 = 1 and 3 / 2 = 1). All even numbers are eliminated on the first try, at i = 2. 9 and 15 are eliminated on the second try, at i = 3. For 17 and 19, 2, 3, and 4 are tried; none divides. The program prints:

```
2 3 5 7 11 13 17 19
```

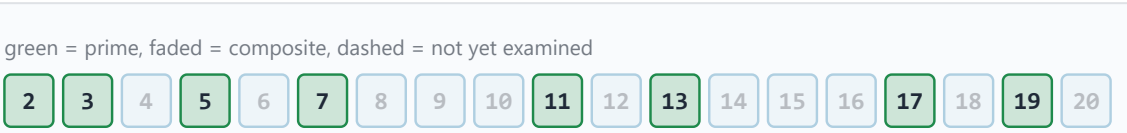


Figure 8.18: The status of the numbers from 2 to 20. The green boxes are the primes: 2, 3, 5, 7, 11, 13, 17, 19.

There are 8 primes between 2 and 20, and 25 between 2 and 100. When many primes are needed, faster methods such as the sieve of Eratosthenes are used instead of testing each number separately. When this method finds a prime, it marks all its multiples in a single pass.

8.8.5 The Fibonacci sequence

Definition 8.14

The **Fibonacci sequence** is the sequence that starts with 0 and 1 and in which every following term is the sum of the two terms before it: 0, 1, 1, 2, 3, 5, 8, 13, ...

To produce the sequence it is enough to keep only the last two terms. a holds the current term and b the one after it. In every iteration a is printed, the next term is computed, and the two variables are "shifted" one step:

```
int terms = 5;
long a = 0, b = 1;
for (int i = 0; i < terms; i++) {
    System.out.print(a + " ");
    long next = a + b;
    a = b;
    b = next;
}
```

i	a printed	next = a + b	New a	New b
0	0	1	1	1
1	1	2	1	2
2	1	3	2	3
3	2	5	3	5
4	3	8	5	8

Table 8.23: The first 5 Fibonacci numbers.

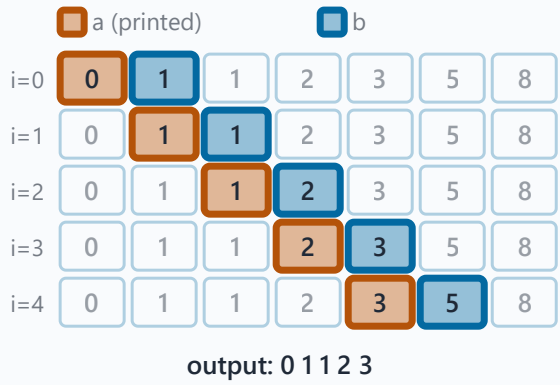


Figure 8.19: The sliding window. In every iteration a (orange) and b (blue) slide one step to the right; a takes the values 0, 1, 1, 2, 3 in turn.

The output is `0 1 1 2 3`. The order of the shift matters: first `a = b`, then `b = next`. If the order is reversed (first `b = next`, then `a = b`), a gets the new value of b, not its old value. The two variables are always equal and every term comes out twice the previous one: `0 1 2 4 8`. The variable `next` exists for exactly this reason; it holds the new value without losing the old one.

The sequence grows exponentially. That is why the variables were chosen as long. Even long is not infinite. The 93rd term, 7,540,113,804,746,346,429, fits in a long. The 94th term exceeds the largest value of long (9,223,372,036,854,775,807); the program prints it as a negative number.

8.9 Loops that work with input

In the loops so far, the number of iterations depended on numbers inside the program. In loops that read input from the user, the user decides when the loop ends. In these loops the number of iterations is not known in advance, and the natural choice is while or do-while.

8.9.1 A number guessing game

The computer keeps a secret number between 1 and 100. The user guesses; the program guides by saying “Higher” or “Lower”. When the guess is right, the number of the try is printed.

```
Random rnd = new Random();
Scanner in = new Scanner(System.in);
int target = rnd.nextInt(100) + 1;  // 1..100
int guess = 0, tries = 0;
while (guess != target) {
  guess = in.nextInt();
  tries++;
  if (guess < target) System.out.println("Higher");
  if (guess > target) System.out.println("Lower");
}
System.out.println("Got it in " + tries + " tries");
```

The `java.util.Random` class is used to generate random numbers; two import lines, for `Random` and `Scanner`, are needed at the top of the file. `rnd.nextInt(100)` gives a number from 0 to 99; adding 1 makes the range 1..100. A different target comes out on every run.

The reason for giving `guess = 0` before the loop is the condition. Because the target is between 1 and 100, 0 can never be the target. So `guess != target` is certainly true at the first check and the loop is entered. The same job could also be written with do-while; in that case this artificial starting value would not be needed.

In a run in which the target is 37, let the user enter 50, 25, and 37 in turn. Table 8.24 shows the three iterations of the game. The table was obtained on JDK 21 by giving the random number generator a seed that yields 37.

Iteration	guess	tries	Message	Possible range
1	50	1	Lower	1...49
2	25	2	Higher	26...49
3	37	3	none	found

Table 8.24: The number guessing game, target = 37, guesses 50, 25, 37.

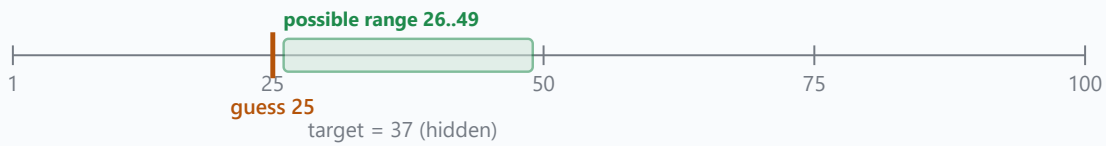


Figure 8.20: After the second guess, the possible range is 26..49. Every guess narrows the range in which the target can be found.

The first guess is 50. Because *50 > 37*, “Lower” is printed; the target is now in the range 1..49. The second guess is 25; “Higher” is printed and the range drops to 26..49. The third guess is 37. Both ifs are false and no message is printed. The condition *37 != 37* comes out false and the loop ends. The program prints “Got it in 3 tries”.

Always guessing the middle of the range roughly halves the range. A range of 100 numbers shrinks to a single number in 7 halvings; that is why with this strategy every target is found in at most 7 tries. This idea is the basis of binary search.

8.9.2 Reading with a sentinel value

If the user does not say in advance how many numbers they will enter, how can the end of the data be recognized? The common solution is to use a special value that cannot be data as the “finished” marker.

Definition 8.15

A **sentinel value** is a special value that marks the end of a data sequence and does not itself take part in the data. A sentinel value must be a value that real data can never take.

```
Scanner sc = new Scanner(System.in);
int sum = 0;
while (true) {
    int num = sc.nextInt();
    if (num == 0) break;
    sum += num;
}
System.out.println("Sum: " + sum);
```

The loop is deliberately written as infinite: `while (true)`. The exit is provided by the break in the middle of the body. As soon as a number is read, it is checked whether it is the sentinel; if it is not, it is added to the sum. Table 8.25 shows the iterations for the input 5, 3, 0.

num read	num == 0	sum
5	`false`	5
3	`false`	8
0	`true`	8 (break)

Table 8.25: Reading with a sentinel value, input 5, 3, 0.

The program prints “Sum: 8”. The 0 has not been added to the sum. If the user enters 0 as the first number, the loop ends in the first iteration and “Sum: 0” is printed. For the input 7, −2, 4, 0 the sum is 9; negative numbers are valid data in this program, because the sentinel is only 0.

Common mistake

Checking the sentinel after processing, or choosing a value that could be data as the sentinel. If the line `sum += num;` is written before the if, the sentinel is also added. While the sentinel is 0 this makes no difference, but if the sentinel were −1, the sum would come out 1 short. In a program such as a grade average, however, 0 can be a real grade. If 0 is chosen as the sentinel, the grade of a student who got 0 is interpreted as “the data is finished”. In such cases a sentinel outside the data range, such as −1, is chosen.

A typical extension of reading with a sentinel is computing an average. Besides the sum, a counter is kept, and after the loop the average is computed as *sum / count*. Two details must not be forgotten: if no numbers were entered (count = 0), no division is done, and a `(double)` cast is made before the division. For the input 80, 64, 90, 0 the sum is 234 and the average is 78.0; for 6, 9, 0 the average is 7.5. Without the cast, integer division would give *15 / 2 = 7*.

8.10 Nested loops

8.10.1 Rows and columns

The body of a loop can contain another loop. This structure is called a **nested loop**. The outer one is called the outer loop and the inner one the inner loop. Tables, grids, multiplication tables, and screen patterns have a row-and-column layout; the natural tool for such output is a nested loop.

```
for (int row = 0; row < 3; row++) {
  for (int col = 0; col < 4; col++) {
    System.out.printf("(%d,%d) ", row, col);
  }
  System.out.println();
}
```

```
(0,0) (0,1) (0,2) (0,3)
(1,0) (1,1) (1,2) (1,3)
(2,0) (2,1) (2,2) (2,3)
```

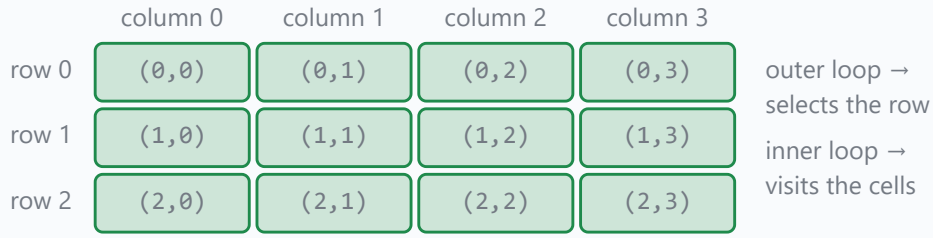


Figure 8.21: A 3 × 4 grid. The outer loop selects the row and the inner loop visits the cells of that row from left to right.

The basic rule of a nested loop is this: the inner loop runs from start to finish in *every* iteration of the outer loop. When row = 0, the inner loop runs four times for col = 0, 1, 2, 3 and prints four cells from (0,0) to (0,3). When col becomes 4, the inner loop ends. Then `println()` ends the line. The outer loop moves to 1 with row++ and the inner loop starts *again* from col = 0. Because the variable col is defined in the header of the inner loop, it is created anew in every outer iteration and gets the value 0.

The inner body runs a total of *number of rows × number of columns = 3 × 4 = 12* times. The `%d` placeholders in `printf` are filled in order with the values given after the comma. `println()`, however, is in the body of the outer loop but *outside* the inner loop. This position makes the output move to the next line once at the end of each row.

The limit of the inner loop can depend on the variable of the outer loop. In that case the number of times the inner body runs is a sum rather than a product. For example, if i runs from 1 to 4 and j from i to 4, the inner body runs *4 + 3 + 2 + 1 = 10* times. The growing triangle pattern is an example of this structure.

8.10.2 The multiplication table

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        System.out.printf("%4d", i * j);
    }
    System.out.println();
}
```

```
   1   2   3
   2   4   6
   3   6   9
```



Figure 8.22: A 3 × 3 multiplication table. i × j is written into cell (i, j); `%4d` right-justifies each number in a field of 4 characters.

When the outer loop has i = 1, the inner loop prints 1, 2, 3 for j = 1, 2, 3. At j = 4 the inner condition becomes false and println ends the line. The same job is repeated for i = 2 and i = 3. The format `%4d` aligns the numbers in columns: three spaces are added in front of one-digit numbers and two in front of two-digit numbers. When the limits are made 10, the classic 10 × 10 multiplication table comes out; because the largest number is 100, a field of 4 characters is still enough.

8.11 Patterns

Screen patterns made of stars or numbers are a classic subject of nested-loop exercises. The patterns themselves are not important; what matters is expressing the layout of an output in numbers and turning those numbers into loop limits. This skill is also needed when working with tables, reports, and matrices.

Definition 8.16

A **pattern** is output written line by line in which the number of spaces and characters in each line changes regularly with the line number. The outer loop produces the lines, and the inner loops produce the space and character parts of a line.

8.11.1 A method for solving pattern problems

A pattern is turned into code in four steps:

1. **Draw.** The expected output is drawn on graph paper for a small n (for example n = 4). Each square is one character; spaces count as squares too.
2. **Count.** The numbers of spaces and characters in each row are written in a table.
3. **Relate.** The numbers are related to the row number. For example, the sequence 3, 2, 1, 0 is n − row, the sequence 1, 2, 3, 4 is row, and the sequence 1, 3, 5, 7 is 2 × row − 1.
4. **Code.** The outer loop runs as many times as there are rows. For each part of a row (spaces, stars) an inner loop is written; the limit of the inner loop is the formula found in step 3. At the end of the row, println is called.

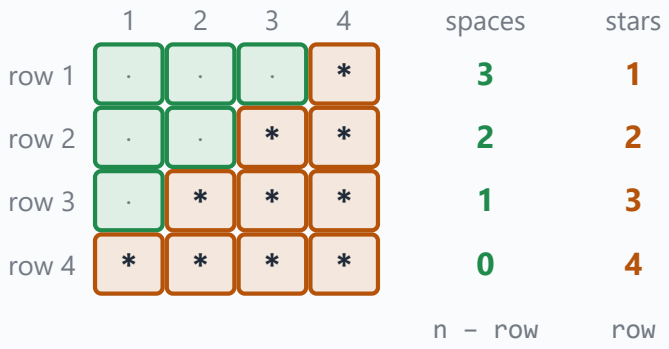


Figure 8.23: The n = 4 right-aligned triangle on graph paper. The space column is 3, 2, 1, 0 (n − row) and the star column is 1, 2, 3, 4 (row).

Table 8.26 collects the formulas of four basic patterns. The numbers on the right were obtained by actually producing the patterns for n = 4 with JDK 21 and counting the spaces and stars in each row.

Pattern	Spaces	Characters
Growing triangle	0	row
Shrinking triangle	0	n − row + 1
Right-aligned	n − row	row
Pyramid	n − row	2 × row − 1

Table 8.26: Numbers of spaces and characters by row.

Row	Growing	Shrinking	Right-aligned	Pyramid
1	0 + 1	0 + 4	3 + 1	3 + 1
2	0 + 2	0 + 3	2 + 2	2 + 3
3	0 + 3	0 + 2	1 + 3	1 + 5
4	0 + 4	0 + 1	0 + 4	0 + 7

Table 8.27: The number of “spaces + stars” in each row for n = 4.

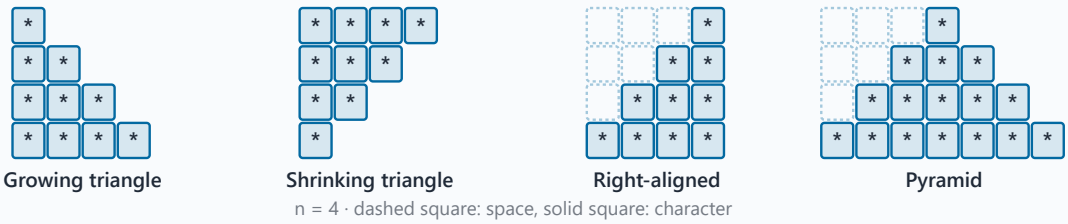


Figure 8.24: Four patterns for n = 4: growing triangle, shrinking triangle, right-aligned triangle, and pyramid. Dashed squares are spaces and solid squares are characters.

8.11.2 The growing triangle and the number triangle

In the growing triangle, row r has r stars and no spaces. A single inner loop is enough; its upper limit is the variable of the outer loop:

```
int n = 4;
for (int row = 1; row <= n; row++) {
    for (int col = 1; col <= row; col++) {
        System.out.print("*");
    }
    System.out.println();
}
```

```
*
**
***
****
```

When row = 1, the inner loop runs only for col = 1 and prints one star. At col = 2, *2 <= 1* is false and println moves to the next line. When row = 2, the inner loop runs twice, when row = 3 three times, and when row = 4 four times. At row = 5 the outer condition becomes false. A total of *1 + 2 + 3 + 4 = 10* stars are printed. In general, a growing triangle of n rows has $n(n + 1) / 2$ stars; for n = 4, $4 \cdot 5 / 2 = 10$.

The same skeleton can print numbers instead of characters. In the number triangle, the variable of the inner loop is written into each cell:

```
int n = 4;
for (int row = 1; row <= n; row++) {
    for (int value = 1; value <= row; value++) {
        System.out.print(value + " ");
    }
    System.out.println();
}
```

```
1
1 2
1 2 3
1 2 3 4
```

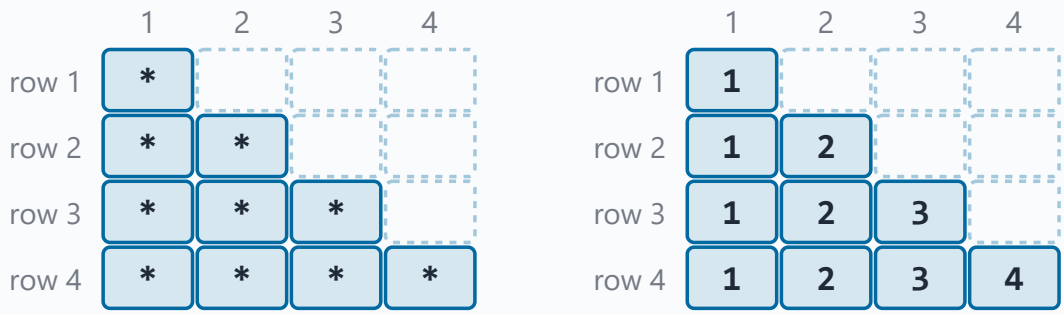


Figure 8.25: The growing triangle and the number triangle for n = 4. The structure of the two patterns is the same; only the value written into the cell changes.

Again the outer loop determines the row and the upper limit of the inner loop determines the number of elements in the row. The difference is only inside the `print` call: `value + " "` is written instead of `"*"`. Because a space is left after every number, there is also a space at the end of the rows; it is not visible on the screen. If the variable of the outer loop had been written instead of the variable of the inner loop, the same number would be repeated in every row: 1, 2 2, 3 3 3, 4 4 4 4.

8.11.3 Patterns with spaces: the right-aligned triangle and the pyramid

In the right-aligned triangle, each row consists of two parts: first spaces, then stars. While the number of spaces decreases, the number of stars increases. A separate inner loop is written for each part:

```
int n = 3;
for (int row = 1; row <= n; row++) {
    for (int space = 0; space < n - row; space++)
        System.out.print(" ");
    for (int star = 0; star < row; star++)
        System.out.print("*");
    System.out.println();
}
```

```
  *
 **
***
```

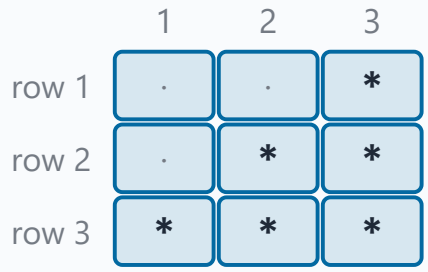


Figure 8.26: The n = 3 right-aligned triangle. Row 1 has 2 spaces and 1 star; row 3 has no spaces and 3 stars.

For n = 3, at row = 1 the space loop runs *n − row = 2* times and the star loop once. At row = 2, 1 space and 2 stars are printed. At row = 3, the condition of the space loop *0 < 3 − 3* is false from the start; the loop never runs and the row begins directly with 3 stars. Every row is exactly n = 3 characters. This check shows that the space and star formulas are consistent with each other: *(n − row) + row = n*.

The pyramid is a variation of the same skeleton. The number of spaces is again n − row, and the number of stars is 2 × row − 1. The stars grow by two in each row; the spaces shrink by one and the group of stars stays in the middle. The following program draws first the pyramid and then, for comparison, the shrinking triangle, for n = 4:

```
public class Pyramid {
    public static void main(String[] args) {
        int n = 4;
        for (int row = 1; row <= n; row++) {
            for (int sp = 0; sp < n - row; sp++) System.out.print(" ");
            for (int st = 0; st < 2 * row - 1; st++) System.out.print("*");
            System.out.println();
        }
        for (int row = 1; row <= n; row++) {
            for (int st = 1; st <= n - row + 1; st++) System.out.print("*");
            System.out.println();
        }
    }
}
```

```
   *
  ***
 *****
*******
****
***
**
*
```

The first four lines of the output are the pyramid. The last four lines are the shrinking triangle that follows in the same program: the number of stars is *n − row + 1*, that is, 4, 3, 2, 1. The total number of stars in the pyramid is *1 + 3 + 5 + 7 = 16 = 4^2*. The sum of the first n odd numbers is always n^2.

8.11.4 The hollow square

In some patterns, what is written in a cell depends not only on the row but on both the row and the column. In the hollow square, the cells on the border are stars and the inner cells are spaces. An if-else goes into the body of the inner loop:

```
int n = 4;
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    if (i == 0 || i == n - 1 || j == 0 || j == n - 1)
      System.out.print("*");
    else
      System.out.print(" ");
  }
  System.out.println();
}
```

```
****
*  *
*  *
****
```

A cell is a star if it is on one of the four borders: the first row (`i == 0`), the last row (`i == n - 1`), the first column (`j == 0`), or the last column (`j == n - 1`). The four conditions are joined with `||`; one being true is enough. Because of short-circuiting, the conditions after the first true one are not evaluated. Table 8.28 shows which condition decides in each cell.

i	j = 0	j = 1	j = 2	j = 3
0	* (i == 0)	* (i == 0)	* (i == 0)	* (i == 0)
1	* (j == 0)	spaces	spaces	* (j == n - 1)
2	* (j == 0)	spaces	spaces	* (j == n - 1)
3	* (i == n - 1)	* (i == n - 1)	* (i == n - 1)	* (i == n - 1)

Table 8.28: The hollow square (n = 4); the first condition that comes out true in each cell.

12 of the 16 cells are border cells and 4 are inner cells. Writing the space in the else branch is mandatory. If this line is left out, nothing is written for the inner cells. The star on the right border shifts left and the middle rows appear as `**`; the square is ruined.

8.11.5 Common mistakes in patterns

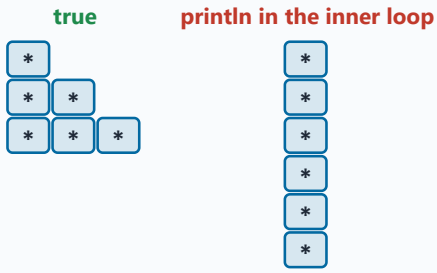


Figure 8.27: On the left, the correct growing triangle. On the right, when println is written in the inner loop, every star falls on a separate line.

Common mistake

Putting the `println` call inside the inner loop. In that case the line ends after every star. For n = 3, instead of the growing triangle, *1 + 2 + 3 = 6* lines come out one under the other, with a single star in each. println must be written after the closing brace of the inner loop, inside the outer loop.

Other typical mistakes in patterns are these. If `<` and `<=` are confused, one character too few or too many comes out in every row; if the counter starts at 0, `<` is used, and if it starts at 1, `<=`. If the space loop is written after the star loop, the spaces fall at the end of the row, are invisible, and the pattern is left-aligned. If the variable of the outer loop is changed by mistake in the inner loop (for example, if `row++` is written in the inner header), rows are skipped or the loop never ends. Every pattern is protected from these mistakes by checking it row by row with a trace table for a small n.

8.12 Test design

That a loop works correctly with a single input does not show that it will work correctly with every input. Loop errors usually appear at the boundaries: in cases where it should not run at all, run one iteration, or process the largest value. A good test set contains an example of each of these cases.

Definition 8.17

A **test case** is a pair of an input chosen for a program and the output expected for that input. An **edge case** is a value at the end of the input range or one that requires special behavior: 0, 1, negative numbers, and the largest or smallest value of a type.

A test set is prepared with these steps:

1. For each problem an ordinary, “normal” value is chosen.
2. 0, 1, negative values, and values close to the upper limit are chosen separately.
3. For each input, the expected result is computed by hand, without running the code.
4. The program is run. If the result differs from the expected one, the variables are traced iteration by iteration and the iteration in which the first deviation occurs is found.

Four tests are a sufficient start for the digit-count program. Table 8.29 gives these tests and the results obtained with JDK 21.

Digit-count test	number	expected	JDK 21 result
Normal	4096	4	4
Zero	0	1	1
Negative	−507	3	3
Extreme value	−2147483648	10	10

Table 8.29: Digit-count tests (the do-while solution with a long copy).

Each test is chosen to catch a specific mistake. The test for 0 gives the result 0 if the loop was written with while, and reveals the mistake. The negative test reveals that the loop never runs if the sign has not been dropped. The extreme-value test gives a wrong result (1) if the copy is kept as an int. The normal test verifies the basic logic of the code. A solution that passes all four tests contains none of these mistakes.

The same thinking applies to every loop. A sum loop is tried with n = 0 (no iterations), n = 1 (a single iteration), and a large n. A prime test is tried with 0, 1, 2, 4, a square number (for example 25), and a large prime. A pattern program is also run with n = 1 and n = 0. Each of the cases in which the loop runs zero times, once, and many times must be covered by at least one test.

8.13 Summary

- Every loop consists of four parts: initialization, condition, body, and update. The update brings the condition closer to its end; if it is forgotten, the loop never ends.
- In the `for` loop, the three parts are gathered in the header and the order is “initialization → condition → body → update → condition …”. In a for loop running from 1 to n, the body runs n times and the condition n + 1 times.
- `while` asks the condition before the body and `do-while` after the body. The body of a while may never run; the body of a do-while runs at least once.
- If the number of repetitions is known, for is chosen; while the condition holds, while; if at least one iteration is needed, do-while. The three can be converted into one another; the choice is for readability.

- `break` ends the nearest loop, and `continue` skips only the rest of that iteration. A labeled break leaves nested loops at once.
- An infinite loop arises from an update in the wrong direction or one that is never made. A double counter accumulates rounding error; the loop counter should be an int.
- A loop invariant is a property that stays true at the start of every iteration; when the loop ends it shows that the result is correct. Against boundary errors, the first iteration, the last iteration, and the number of iterations are checked.
- The digit pattern (`% 10` gives the last digit, `/ 10` removes it) is used for digit counts, sums of digits, reversal, palindromes, Armstrong numbers, and changing the base.
- The divisor pattern rests on the check `n % i == 0`. For perfect numbers it is enough to search up to n / 2, and for the prime test up to √n.
- A sentinel value is a value that cannot be data; it marks the end of the data and is not itself processed.
- In a nested loop, the inner loop runs from start to finish in every iteration of the outer loop. In patterns, the outer loop produces the row and the inner loops produce the spaces and characters in the row; the numbers are tied to the row number.
- Loops are tested with normal, 0, 1, negative, and extreme values; the expected result is computed by hand beforehand.

8.14 Key terms

loop loop body iteration initialization condition loop variable scope counter accumulator increment / decrement pre-test loop post-test loop menu input validation break statement labeled break infinite loop overflow floating-point number loop invariant off-by-one error pattern digit integer division palindrome Armstrong number exponent / base perfect number divisor prime number flag short-circuit evaluation Fibonacci sequence random number sentinel value nested loop inner loop / outer loop pattern (output) code tracing trace table test case edge case

---

8.15 Try it yourself

1. What does the loop `for (int i = 2; i <= 20; i *= 3) System.out.print(i + " ");` print?

How many times do the body and the condition run?

Answer

i takes the values 2, 6, 18 in turn and `2 6 18` is printed. After 18, *i = 18 × 3 = 54* and *54 <= 20* is false. The body runs 3 times and the condition 4 times. The update does not have to be an increase by 1; here the counter advances by multiplication.

2. After the loop `int x = 1; while (x < 100) x *= 2;`, what is x? How many iterations does the loop run?

Answer

x becomes 2, 4, 8, 16, 32, 64, 128 in turn. Because 64 < 100 is true, one more iteration runs and x = 128; *128 < 100* is false. The loop runs 7 iterations and x = 128. The loop ends at the first value that makes the condition false; this value may have passed the limit.

3. After `int k = 10; do { k -= 4; } while (k > 0);` runs, what is k? How many times does the body run? What would happen if the same loop were written as `while (k > 0) k -= 4;` with the starting value k = −1?

Answer

k becomes 6, 2, −2 in turn. The condition is asked at the end of each iteration; it is true for 6 and 2 and false for −2. The body runs 3 times and k = −2. In the while loop starting with k = −1, *−1 > 0* is false from the start; the body never runs and k stays −1. With the same starting value, a do-while would run once and k would be −5.

4. What does the following loop print? `for (int i = 1; i <= 10; i++) { if (i % 2 == 0) continue; if (i > 7) break; System.out.print(i + " "); }`

Answer

Even numbers are skipped with continue. Of the odd numbers, 1, 3, 5, and 7 are printed. At i = 9, 9 is odd, the first condition is false, *9 > 7* is true, and break ends the loop. The output is `1 3 5 7`. Because i = 8 is skipped with continue, break gets its turn at 9, not at 8.

5. With the digit pattern, what are the sum of digits and the reverse of 4096? What do the reverse of

1200 and the palindrome test give? What do the reverse and the palindrome test give for 0?

Answer

4096: the digits from right to left are 6, 9, 0, 4. The sum is *6 + 9 + 0 + 4 = 19* and the reverse is 6904; because 6904 ≠ 4096, it is not a palindrome. 1200: the digits are 0, 0, 2, 1 and the reverse is built as *0 → 0 → 2 → 21*; the result is 21 and it is not a palindrome. For 0 the while loop never runs; rev stays 0 and because *0 == 0*, the test gives `true`. 0, as a one-digit number, really is a palindrome.

6. After the following nested loop, what is count? `for (int i = 1; i <= 4; i++) for (int j = i + 1; j <= 4; j++) count++;` (count is 0 at the start)

Answer

For i = 1, j = 2, 3, 4 (3 iterations); for i = 2, j = 3, 4 (2 iterations); for i = 3, j = 4 (1 iteration); for i = 4 the inner loop never runs (*5 <= 4* is false). The total is *3 + 2 + 1 + 0 = 6*. This number is the number of different pairs that can be chosen from 4 elements: *4 · 3 / 2 = 6*.

7. The following pattern is wanted for n = 4. What are the numbers of spaces and stars in row r? What should the limits of the inner loops be?

```
****
 ***
  **
   *
```

Answer

Row 1 has 0 spaces and 4 stars, row 2 has 1 space and 3 stars, row 3 has 2 spaces and 2 stars, and row 4 has 3 spaces and 1 star. The number of spaces is *r − 1* and the number of stars is *n − r + 1*. Every row is *(r − 1) + (n − r + 1) = n* characters. Code: `for (int space = 0; space < row - 1; space++)` for the spaces, `for (int star = 0; star < n - row + 1; star++)` for the stars.

8. Does the loop `for (double d = 0; d != 1.0; d += 0.25)` end? How many iterations does it run? What would happen if the same loop were written with `d += 0.1`?

Answer

0.25 = 1/4 is represented exactly in binary (0.01_2). d becomes 0, 0.25, 0.5, 0.75, and exactly 1.0 in turn. The loop runs 4 iterations and ends. 0.1, however, cannot be represented exactly in binary; after 10 additions d = 0.9999999999999999 and in no iteration is it exactly equal to 1.0. With the `!=` condition this loop never ends. The safe way is an int counter.

9. The prime test (with the condition `prime && i <= n / i`) is run for n = 97. How many divisor candidates are tried? At which value of i and why does the loop end?

   Answer

   i = 2, 3, ..., 9 are tried; none divides 97. That is 8 candidates. At i = 9, *9 <= 97 / 9 = 10* is true. At i = 10, *10 <= 97 / 10 = 9* is false and the loop ends. Because √97 ≈ 9.85, there is no need to look for a larger divisor. The program prints `true`.

8.16 Further reading

- R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), Section 1.3: conditionals and loops in Java, explained with trace tables and short programs such as powers, harmonic numbers, Newton's method, and binary representation; the online version of the section is at introcs.cs.princeton.edu.
- A. B. Downey, C. Mayfield, *Think Java* (2nd ed., O'Reilly, 2019): while and for loops and nested loops explained plainly for beginners; the book's online version is free.
- Oracle, *The Java Tutorials*: the pages while and do-while, for, and branching statements; the official examples of labeled break and continue are on these pages.

8.17 References

1. P. Deitel, H. Deitel, *Java: How to Program, Early Objects*, 11th ed., Pearson, 2018.
2. Y. D. Liang, *Introduction to Java Programming and Data Structures*, 13th ed., Pearson, 2024.
3. Oracle, *The Java Tutorials: while and do-while, for, Branching Statements*.
4. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 1.3 Conditionals and Loops.
5. A. B. Downey, C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019.

Methods

As a program grows, the same jobs start to be written again and again in different places, and the `main` method becomes too long to read. A method gathers a task under a name. The task is written once and then called by name wherever it is needed. These notes first show how a method is defined and called. Next comes what happens in memory during a call: arguments being copied into parameters, the frame opened for each call, and the call stack these frames form. Return values, `void` methods and early return are treated separately. Why Java always passes arguments by value is shown step by step with the swap example. The real error messages given by the compiler for overloading, variable-length arguments, scope and shadowing are also examined. The last sections cover good method design and example methods such as even/odd, digit count, primality, GCD and LCM. All the programs in the notes were compiled and run with JDK 21.

LEARNING OUTCOMES

Defining
Write the header and body of a method; call a method from `main` and from other methods; explain the signature of the `main` method and the meaning of the `static` modifier.

Data flow
Tell a parameter from an argument; use a return value in an expression; use a `void` method and early return in the right places; show by tracing that Java passes arguments by value; work with overloading and variable-length arguments.

Scope
Explain where a local variable is visible and how long it lives; trace the frames on the call stack step by step; recognize shadowing and scope-related compile errors.

Design
Split a problem into single-purpose methods; write a method's contract; build short, focused, testable methods that return their result instead of printing it.

9.1 What is a method?

In the Java programs written so far, all the work was done in one place, inside the `main` method. In short programs this is enough. But when a program reaches a few hundred lines, the same computation gets rewritten in three or four places. When a bug is fixed, the other copies are forgotten. What `main` does, and in what order, also becomes unreadable. Methods are the solution to this problem.

9.1.1 Naming a task

In mathematics, a function produces an output for its input: for $f(x) = x^2$, $f(3) = 9$. A function in programming is a similar unit. A piece of code that does a particular task is given a name. This code takes its inputs from outside, does its job and, if necessary, gives a result. Since everything in Java is written inside a class, functions are defined inside a class too, and they are called **methods**.

Definition 9.1

A **method** is a named group of statements defined inside a class that does a particular task. A method receives its inputs through its **parameters** and gives its result as its **return value**. A method is run by writing its name followed by values in parentheses; this is called a **method call**.

The methods in Java's own library illustrate this idea well. The call `Math.sqrt(16)` gives the square root of 16, that is, `4.0`. You do not need to know how the square root is computed. Three things need to be known: the name of the method (`sqrt`), what it asks for (a number) and what it gives (the square root). The internal details are hidden from the user of the method. Figure 9.1 shows this as a black box.

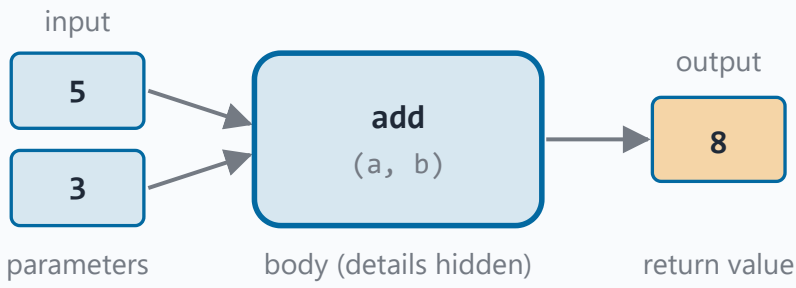


Figure 9.1: A method is like a black box. `add` takes two inputs (5 and 3) and gives one result (8). The inputs enter through the parameters and the result leaves as the return value; the details in the body are not visible from outside.

The box in the figure has two inputs on its left and one output on its right. What is inside the box can be seen only by looking at the body of the method. The code that calls the method thinks only "give it these two numbers and get the sum back". This separation is the most important gain of methods: *what* a task does is separated from *how* it does it.

9.1.2 Why write methods?

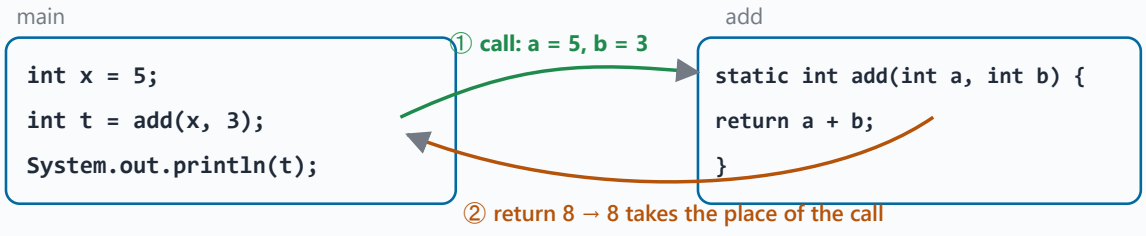
Suppose a student is writing a program that prints the grades of three courses on the screen. The student wants to draw a line before and after each course. Without methods, the loop that draws the line is copied six times. If the length of the line were to change, all six places would have to be corrected; if one is forgotten, the output is inconsistent. If this loop is put into a method named `drawLine`, a one-line call remains in each of the six places: `drawLine(5);`. If the length changes, only the argument changes.

The gains of writing methods can be listed as follows:

- **Avoiding repetition.** The same code is written once. When a bug is found, it is fixed in one place and the fix applies to every call.
- **Readability.** A well-named method call says what is being done. The line `if (isPrime(n))` is much easier to read than a loop looking for divisors.
- **Testability.** A small method can be tried with a few sample inputs independently of the rest of the program. That `digitCount(-507)` gives 3 can be checked in a single line.
- **Reuse.** A method written once is used in other programs and inside other methods. A method that computes the LCM shortens its work by calling the GCD method.
- **Division of labor.** A big problem is split into small subtasks. Each subtask becomes a separate method and is written and tried separately.

9.1.3 What happens in a call?

When a method is called, the flow of the program temporarily leaves the place that made the call, goes into the body of the method, and returns to where it left off when the method ends. Figure 9.2 shows this round trip.



③ main continues where it left off: t = 8, then println

Figure 9.2: The journey of a call. (1) `main` calls `add(x, 3)`; control passes to the method and a = 5, b = 3. (2) `return` sends 8 back; 8 takes the place of the call. (3) `main` carries on where it left off: t becomes 8, then

`println` runs.

This journey consists of the same five steps in every call:

1. **The arguments are evaluated.** The values of the expressions inside the parentheses of the call are found. In `add(x, 3)`, the current value of x (5) is read.
2. **The values are copied into the parameters.** A new area of memory is opened for the method. The first value is written into the first parameter and the second value into the second parameter: a = 5, b = 3.
3. **The body runs.** The statements between the curly braces of the method run in order.
4. **A value is returned.** When the `return` statement is reached, the method ends and the value next to `return` is sent to the place that called it.
5. **The calling code continues.** The returned value takes the place of the call expression. The program continues from the line where the call is.

One point in these steps must not be overlooked: each call works with its own parameters and its own local variables. If the same method is called twice, two separate areas of memory are opened. The second call does not see the variables of the first; when the first call ends, its variables have already been deleted. Pass by value, scope and the call stack all rest on this rule.

9.2 Method definition and call

To use a method, you must first *define* it, that is, write its name, its inputs, the type of its result and its body. A defined method is then *called* wherever you like and as many times as you like. The definition is made once, the call many times.

9.2.1 The first method: add

The `add` method, which gives the sum of two integers, and the two lines that use it, look like this:

```
static int add(int a, int b) {
    int sum = a + b;
    return sum;
}

int total = add(5, 3);
System.out.println(total);
```

The first four lines are the definition. The line `static int add(int a, int b)` is the *header* of the method. `int` says that the method will return an integer. `add` is the name of the method. The `int a, int b` inside the parentheses declare that the method expects two integers and that they will be

reached inside the method under the names `a` and `b`. The two lines between the curly braces are the *body* of the method: first a + b is written into a variable named `sum`, and then this value is given back with `return`. The last two lines are the call: `add(5, 3)` runs the method with 5 and 3, and the returned value is assigned to `total` and printed.

These fragments do not compile on their own; they must be inside a class. The complete program is below. The method is written inside the same class as `main`, outside `main`. Methods cannot be defined inside one another: writing `add` inside the curly braces of `main` is a compile error.

```
public class Add {
    static int add(int a, int b) {
        int sum = a + b;
        return sum;
    }

    public static void main(String[] args) {
        int total = add(5, 3);
        System.out.println(total);
    }
}
```

```
8
```

The program always starts running from `main`. Even if `add` is written before `main` in the class, it does not run on its own; it runs only when it is called. The order of methods within the class does not matter. `add` could just as well have been written below `main`.

The `a` and `b` in the definition and the `5` and `3` in the call play different roles and have different names.

Definition 9.2

A **parameter**, or more fully a **formal parameter**, is a variable declared with its type in the method header; it says which inputs the method expects. An **argument**, or **actual argument**, is a value written inside the parentheses at the time of the call and copied into the parameter.

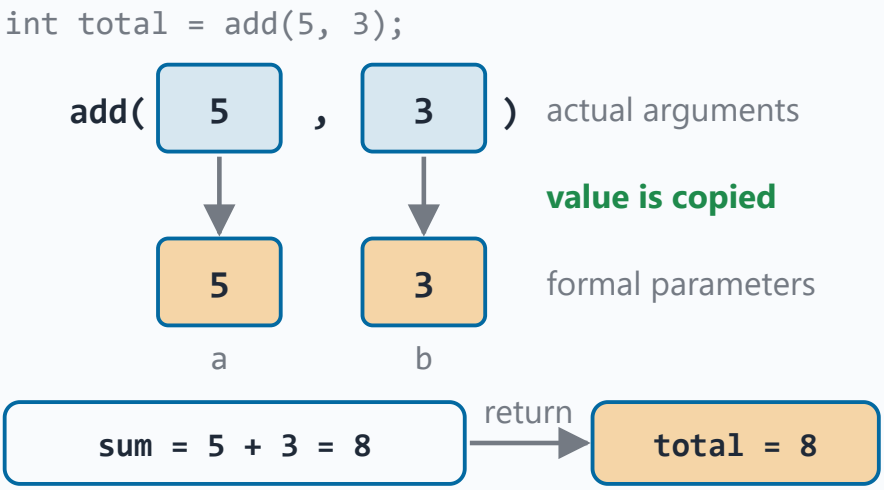


Figure 9.3: In the call `int total = add(5, 3);`, the actual arguments (5 and 3) are copied into the formal parameters (a and b). In the body sum = 5 + 3 = 8 is computed, and `return` sends 8 to `total`.

A parameter is a variable; an argument is a value, or an expression whose value is to be computed. Parameters come to life anew each time the method is called and are filled with the arguments of that call. Arguments are matched to parameters *in order*: the first argument to the first parameter, the second argument to the second. The number of arguments must equal the number of parameters, and the type of each argument must fit the corresponding parameter. The calls `add(5)` and `add(5, 3, 1)` do not compile, because `add` expects exactly two integers.

```
public class MissingArgument {
    static int add(int a, int b) { return a + b; }
    public static void main(String[] args) {
        System.out.println(add(5));
    }
}
```

```
MissingArgument.java:4: error: method add in class MissingArgument cannot be applied to
given types;
        System.out.println(add(5));
                           ^
  required: int,int
  found:    int
  reason: actual and formal argument lists differ in length
1 error
```

This time the compiler gives a longer message: the method `add` cannot be applied to the given types. The line `required: int,int` shows the parameter types the method expects, and `found: int` shows the arguments found in the call. The last line gives the reason: the actual and formal argument lists differ in length.

Common mistake

Rewriting the type of the argument in the call: `add(int 5, int 3)`. Types are written only in the definition, next to the parameters. In a call there are only values or expressions: `add(5, 3)`. Conversely, forgetting the parameter type in the definition is also an error: `static int add(a, b)` does not compile. Each parameter's own type is written separately; the form `int a, b` is not valid in a parameter list.

9.2.2 Using the result of a call

The call of a method that returns a value is an *expression* in Java. After the call runs, the returned value takes its place. So a call can be used anywhere a value can be used: on the right of an assignment, inside an arithmetic operation, in a condition, or as an argument to another method. The following program shows four different uses together:

```
public class UseAdd {
    static int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        int x = 5;
        int t = add(x, 3);                    // a variable and a constant as arguments
        System.out.println(t);
        System.out.println(add(10, 20) * 2);   // the call is part of an expression
        System.out.println(add(add(1, 2), 4)); // the result of a call is an argument to
another call
        add(7, 8);                            // the returned value is not used
    }
}
```

```
8
60
7
```

The first call takes a variable and a constant as arguments. Since the value of x is 5, *add(5, 3) = 8* is computed. In the second call, `add(10, 20)` first returns 30, then this value is multiplied by 2 and 60 is printed. In the third line the result of one call becomes an argument to another call. Java first runs the inner `add(1, 2)` and gets 3; then the outer call runs as `add(3, 4)` and 7 is printed.

The `add(7, 8);` in the last line is also a valid statement. The method returns 15, but since this value is not used anywhere, it is discarded. Java does not require the returned value to be used. However, this line adds nothing to the program: `add` only computes; it does not print anything and does not

change any variable. If a method that returns a value is called and its result is thrown away, this is usually a sign of forgetfulness.

If there are several calls in one expression, Java evaluates the arguments and operands from left to right. The small method below prints its argument in square brackets, so that you can tell it has been called, and returns twice its argument:

```
public class OrderDemo {
    static int twice(int x) {
        System.out.print("[" + x + "] ");
        return 2 * x;
    }

    public static void main(String[] args) {
        int r = twice(3) - twice(1);
        System.out.println(r);
        System.out.println(twice(twice(1)));
    }
}
```

```
[3] [1] 4
[1] [2] 4
```

In the expression `twice(3) - twice(1)`, the call on the left runs first and prints `[3]`, then the one on the right runs and prints `[1]`. The subtraction is done when both values are ready: *6 − 2 = 4*. In the nested call on the last line, the inner `twice(1)` must run first, because it provides the argument of the outer call. `[1]` is printed first, then `[2]`, and 4 is printed. Printing to the screen and returning a value are independent of each other. Whatever the method prints, only its return value takes part in the expression.

9.2.3 add(5, 3) step by step

To see in detail what happens in a call, the code is executed line by line. The line numbers refer to the code below; lines 7 and 8 are inside `main`.

```
static int add(int a, int b) {
    int sum = a + b;
    return sum;
}

// in main:
int total = add(5, 3);
System.out.println(total);
```

Table 9.1 gives, at each step, which line is running, which frames are open in memory and the variables in the topmost frame. A frame is the area of memory set aside for one call; its exact definition is given together with the call stack.

Step	Line	Call stack (top first)	Variables (top frame)	Output
1	7	`main()`	—	—
2	1	`add(5, 3)` ← `main()`	a=5, b=3	—
3	2	`add(5, 3)` ← `main()`	a=5, b=3, sum=8	—
4	3	`add(5, 3)` ← `main()`	a=5, b=3, sum=8	—
5	7	`main()`	total=8	—
6	8	`main()`	total=8	8

Table 9.1: The trace table of the call `add(5, 3)`. In the call stack column the topmost frame is written first.

Execution starts at line 7 in `main`. A value cannot be written into `total` before the right side of the assignment has been computed. Since the right side has a call, that must run first. In step 2 a new frame is opened for `add` and the arguments are copied: a = 5, b = 3. In step 3 the local variable `sum` is created and gets the value 8. This variable lives only in the frame of `add`; `main` cannot see it. In step 4 `return sum;` runs, 8 is sent to the caller and the frame of `add` is closed. In step 5 execution returns to line 7. 8 has taken the place of the call and the assignment is completed: total = 8. At this point a, b and sum no longer exist. In step 6 the 8 is printed.



Figure 9.4: Three moments of the call `add(5, 3)`. (a) The frame of `add` has been opened above `main`: a = 5, b = 3. (b) `return 8`: the frame is closing. (c) Back in `main`: total = 8; a, b and sum no longer exist.

A detail in the trace table deserves attention: line 7 appears twice. The first time is when the call starts, and the second is when the assignment is completed after the call ends. A line being “left unfinished” and then resumed is a natural consequence of method calls. When execution meets a call, it suspends the line it is on, finishes the method and comes back to the same line.

9.3 The call stack

In the `add` example there were only two methods: `main` and `add`. In real programs a method calls another method, and that one calls yet another. At one moment several methods are left “half done”, waiting. Java keeps track of which method is where, and which variables belong to whom, with a stack structure.

9.3.1 Stack and frame

Think of plates stacked on top of one another in a cafeteria. A new plate is always put on top, and a plate is always taken from the top. The last plate put on is the first one taken. This arrangement is called *last in, first out*, and a structure that behaves this way is called a **stack**. Method calls open and close in exactly this order.

> **Definition 9.3**
>
> A **stack frame** is the area of memory set aside for a method call. The frame holds the parameters and local variables of that call and the information about where execution should return when the method ends. The **call stack** is the stack made up of the frames of all the calls currently in progress. Each call puts a new frame on top of the stack; when the method ends, the top frame is removed.

When the program starts, there is a single frame on the stack: the frame of `main`. When `main` calls a method, the frame of that method is put on top of `main`'s. If that method calls another, the stack rises one level further. Two rules follow from this arrangement. First, only the method of the topmost frame runs; the ones below wait for the call above them to finish. Second, the method called last finishes first. A method cannot finish before the method it called finishes.

9.3.2 avg(9, 7) step by step

In the following example, the `avg` method calls the `add` method to find the average of two numbers. So there are three frames on the stack at once:

```
static int add(int a, int b) {
    return a + b;
}
static int avg(int x, int y) {
    int t = add(x, y);
    return t / 2;
}
int o = avg(9, 7);  // in main
System.out.println(o);
```

Line numbers: 1–3 are `add`, 4–7 are `avg`, and 8–9 are the two lines in `main`. Table 9.2 gives the nine steps.

Step	Line	Call stack (top first)	Variables (top frame)	Output
1	8	`main()`	—	—
2	4	`avg(9, 7) ← main()`	x=9, y=7	—
3	5	`avg(9, 7) ← main()`	x=9, y=7	—

Step	Line	Call stack (top first)	Variables (top frame)	Output
4	1	`add(9, 7) ← avg(9, 7) ← main()`	a=9, b=7	—
5	2	`add(9, 7) ← avg(9, 7) ← main()`	a=9, b=7	—
6	5	`avg(9, 7) ← main()`	x=9, y=7, t=16	—
7	6	`avg(9, 7) ← main()`	x=9, y=7, t=16	—
8	8	`main()`	o=8	—
9	9	`main()`	o=8	8

Table 9.2: The trace table of the call `avg(9, 7)`. In steps 4 and 5 there are three frames on the stack.

In step 1, `main` calls `avg(9, 7)`; there is still a single frame on the stack. In step 2 the frame of `avg` is put on top of `main` and the arguments are copied: x = 9, y = 7. In step 3, `avg` itself makes a call on line 5: `add(x, y)`. What is sent is not the variables x and y themselves but their values: 9 and 7. In step 4 the frame of `add` is put on top and there are three frames. At this moment only `add` runs. `avg` waits on line 5, and `main` on line 8.

In step 5, `add` returns 16 and its frame is removed. The frame opened last has closed first. In step 6 execution returns to line 5 of `avg`: 16 takes the place of the call and t = 16. In step 7, `return t / 2;` runs. The quotient of two integers is integer division: *16 / 2 = 8*. The frame of `avg` closes too. In step 8 execution returns to `main` and o = 8. At this moment none of the variables x, y, t, a and b exists any more. In step 9, 8 is printed.

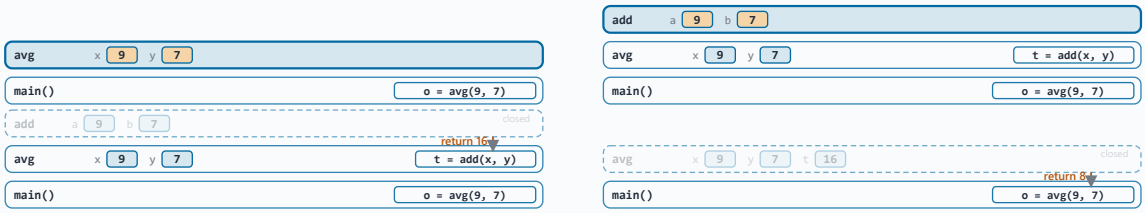


Figure 9.5: Four moments of the call stack. (a) The frame of `avg` is above `main`: x = 9, y = 7. (b) `add` is on top; there are three frames on the stack. (c) `add` returns 16 and closes. (d) `avg` returns 8 and closes; only `main` will remain on the stack again.

The whole program and its real output are below. The call `avg(9, 8)` on the second line shows the effect of integer division: *9 + 8 = 17* and *17 / 2 = 8*. Although the real average is 8.5, the return type is `int` and the division is between two `int`s, so the fraction is discarded.

```java
public class Average {
    static int add(int a, int b) {
        return a + b;
    }

    static int avg(int x, int y) {
        int t = add(x, y);
        return t / 2;
    }

    public static void main(String[] args) {
        int o = avg(9, 7);
        System.out.println(o);
        System.out.println(avg(9, 8));
    }
}
```

```
8
8
```

Common mistake

Taking the same names in different methods to be the same variable. The x of `avg` and an x in `main`, or the a of `add` and the a of another method, are completely independent of one another. Each is a separate box of memory in its own frame. A value passes from one method to another in only two ways: as an argument in a call, or as a return value with `return`.

9.3.3 Rules of the stack

The call stack is not specific to Java; method or function calls in C, Python and almost every programming language work in the same way. If the following rules are kept in mind while tracing, the answer to most questions follows on its own:

- Every call opens a new frame. If the same method is called twice, two separate frames are opened, and the second call does not see the variables of the first.
- Only the topmost frame runs. Every frame below it is in the middle of a call, waiting for the result of that call.
- When a method ends, its frame is deleted with all its contents. Local variables and parameters do not carry values from one call to another.
- The returned value takes the place of the call in the expression where the call was made, and the computation continues from there.

- The last frame opened is the first to close. Frames close in the reverse of the order in which they were opened.

The stack also has a limit. Java sets aside a stack of limited size for each thread. If methods that call one another without end fill the stack, the program stops with a `StackOverflowError`. In ordinary programs the stack depth does not go beyond a few or a few dozen frames. The Method Call Stack simulator animates the opening and closing of frames step by step.

9.4 The parts of a method definition

A method definition consists of two parts: the header and the body. The header is the face of the method that is visible to the outside world. It says who may call it, what it will return, its name and which inputs it expects. The body contains the statements that do the work of the method.

9.4.1 Header and body

Figure 9.6 breaks the header `public static int add(int a, int b)` into its parts.

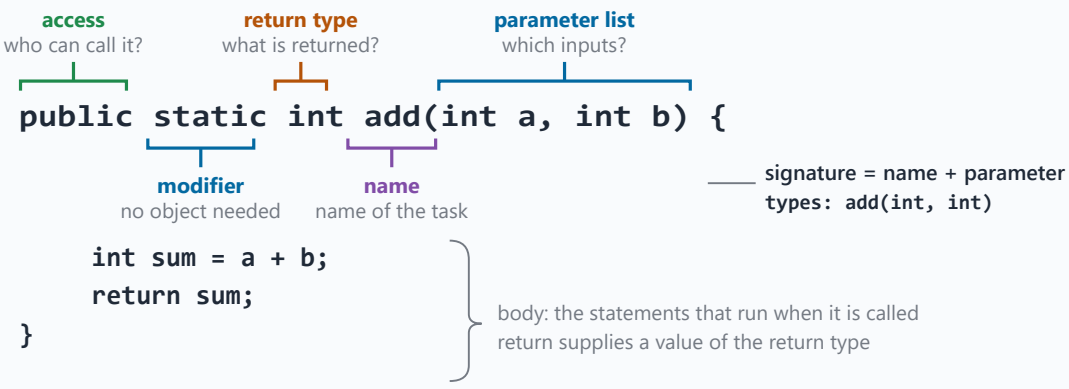


Figure 9.6: The parts of a method definition: the access modifier (`public`), the modifier (`static`), the return type (`int`), the name (`add`), the parameter list (`int a, int b`) and the body between curly braces. The name and the parameter types together form the signature of the method: `add(int, int)` .

Part	Example	Question answered	Explanation
access modifier	`public`	Who can call it?	`public` can be called from anywhere, `private` only from the same class. If nothing is written, classes in the same package can call it.
modifier	`static`	Is an object needed?	A `static` method belongs to the class; no object has to be created to call it.
return type	`int`	What is returned?	The type of the value the method gives. `void` if it does not return a value.

Part	Example	Question answered	Explanation
name	`add`	What is the name of the task?	An identifier that says what the method does.
parameter list	`(int a, int b)`	Which inputs?	"Type name" pairs separated by commas; empty parentheses `()` if there are no inputs.
body	`{ … }`	How is it done?	The statements that run when it is called; `return` supplies a value of the return type.

Table 9.3: The parts of a method header and the questions they answer.

The order of the parts in the header is fixed: first the access modifier and the modifier, then the return type, then the name and the parameter list. An order such as `int static add(…)` does not compile. The access modifier and `static` are optional; the return type, the name and the parentheses are required. In the examples in these notes, methods are written in the same class as `main` and as `static`. The access modifier is mostly left out, because the methods are called only from the same class anyway.

Definition 9.4

The **method signature** is the name of a method together with the list of its parameter types, written in order. The signature of the method `static int add(int a, int b)` is `add(int, int)`. The return type, the parameter names and the modifiers are not part of the signature. A class cannot contain two methods with the same signature.

The signature is the information the compiler looks at when deciding which method to bind a call to. When the compiler sees the call `add(5, 3)`, it looks for a method named `add` that accepts two `int`s. It is natural that the parameter names (`a`, `b`) are not in the signature, because these names never appear in a call. The return type is not in the signature either, because the call itself carries no hint about the type of the returned value.

9.4.2 Method names

Method names follow the same rules as variable names: they start with a letter, may contain letters, digits and underscores, and cannot be one of Java's reserved words. By Java convention, method names start with a lowercase letter, and when several words are joined, the first letter of each later word is capitalized: `digitCount`, `printPositive`, `max3`. Letters outside the basic English alphabet are valid in Java, but they are often avoided so that files open without trouble on different computers.

A good method name says what the method does. Methods that perform an action are named with a verb: `drawLine`, `writeReport`, `readGrades`. Methods that compute a value take the name of that value: `average`, `area`, `gcd`. Methods that return `boolean` read like a question: `isEven`, `isPrime`.

When a method named this way is used in a condition, the line reads almost like plain English: `if (isPrime(n))`, "is n prime?".

9.4.3 The main method and static

The `main` found in every Java program is also a method, and every word in its header has a reason.

Part	Why?
`public`	So that the JVM (Java Virtual Machine) can call the method from outside the class.
`static`	So that the method can be called without creating any object.
`void`	`main` does not return a value.
`main`	The name the JVM looks for in order to start the program.
`String[] args`	The arguments given on the command line when the program is run.

Table 9.4: The parts of the header `public static void main(String[] args)`.

The meaning of the `static` modifier is as follows. Java is an object-oriented language; most methods belong to an object and are called through that object. A `static` method, on the other hand, belongs not to an object but to the class itself. No object has to be created first in order to call it. When the JVM starts the program, there are no objects yet. For it to be able to run `main`, this method must not need an object; that is, it must be `static`.

This has a consequence: a `static` method cannot directly use a non-`static` method or field of the same class, because there is no object to which that method would belong. Below is the example showing this error inside a complete class, together with the message given by the JDK 21 compiler:

```
public class Demo {
    void greet() { }
    public static void main(String[] args) {
        greet();   // error
    }
}
```

```
Demo.java:4: error: non-static method greet() cannot be referenced from a static context
        greet();   // error
        ^
1 error
```

The message says "the non-`static` method `greet()` cannot be referenced from a `static` context". The compiler shows the line of the error (`Demo.java:4`) and its place with the `^` mark. Within the scope of these notes, the fix is simple: make the called method `static` too.

```
public class StaticFix {
    static void greet() {
        System.out.println("Hello");
    }

    public static void main(String[] args) {
        greet();
    }
}
```

```
Hello
```

The parameter `String[] args` of `main` receives the words typed on the command line when the program is run, as an array. The following program prints how many arguments arrived and the arguments themselves.

```
public class MainArgs {
    public static void main(String[] args) {
        System.out.println(args.length);
        for (String a : args) System.out.println(a);
    }
}
```

When run without arguments with the command `java MainArgs`, `args` is an empty array and the program prints only 0. When run with `java MainArgs Ali 42`, the output is:

```
2
Ali
42
```

Each word on the command line arrives as a separate `String`. `42` is not a number but a text of two characters.

Common mistake

Writing the header of `main` incompletely or differently. If `static` is forgotten or the name is written `Main`, the program may compile, but when it is run the JVM cannot find the starting point and does not start the program. Java is case-sensitive: `main` and `Main` are two different names.

9.5 The return value

When a method finishes its work, it can give the caller a result, or it can simply do a job without giving anything back. Which of the two it does is determined by the return type in the header. The result is given with the `return` statement.

9.5.1 The return statement

> **Definition 9.5**
>
> The **return type** is the type of the value the method gives; it is written in front of the name in the header. The **return value** is the value the method sends to the caller with a `return` statement. The statement `return expression;` computes the value of the expression, ends the method at once and **returns** this value to the place where the call was made.

`return` does two jobs at once: it sends a value and it ends the method. Statements written after `return` on the same path do not run. The compiler rejects a statement that comes immediately after `return` and can never be reached by any path with an “unreachable statement” error.

The type of the returned value must match the return type. A method that returns `int` can write `return 8;` but cannot write `return "eight";`. If the return type is `double`, `return 8;` is valid; the `int` value is widened to `double` automatically and `8.0` goes to the caller. The reverse is not valid: a method that returns `int` cannot write `return 2.5;`, because the fractional part would be lost.

9.5.2 void methods

Not every method has to compute a value. A method that draws a line on the screen, writes a report or adds a record to a file has no result to send to the caller. The return type of these methods is `void`. `void` means “empty”: the method does not return a value to the call.

```
static void drawLine(int n) {
    for (int i = 0; i < n; i++)
        System.out.print("-");
    System.out.println();
}
// in main:
drawLine(5);           // -----
int x = drawLine(5);   // ERROR: void has no value
```

The `drawLine` method prints n dashes and ends the line. The method's job is a **side effect**: it changes the state of the screen but gives the caller no value. The complete program and its output:

```java
public class DrawLine {
    static void drawLine(int n) {
        for (int i = 0; i < n; i++)
            System.out.print("-");
        System.out.println();
    }

    public static void main(String[] args) {
        drawLine(5);
        System.out.println("Methods");
        drawLine(8);
        drawLine(0);
        System.out.println("end");
    }
}
```

```
-----
Methods
--------

end
```

`drawLine(5)` prints five dashes and `drawLine(8)` eight. In the call `drawLine(0)` the loop never runs; only `println()` runs, and an empty line appears in the output. The last line in the example, `int x = drawLine(5);`, does not compile. Because the method gives no value, it cannot be used on the right of an assignment:

```
VoidAssign.java:9: error: incompatible types: void cannot be converted to int
        int x = drawLine(5);   // ERROR: void has no value
                         ^
1 error
```

The message says "incompatible types: `void` cannot be converted to `int`". For the same reason, a `void` call cannot be an argument to another method either:

```java
public class VoidPrint {
    static void drawLine(int n) {
        System.out.println("-".repeat(n));
    }
    public static void main(String[] args) {
        System.out.println(drawLine(5));
    }
}
```

```
VoidPrint.java:6: error: 'void' type not allowed here
        System.out.println(drawLine(5));
                                    ^
1 error
```

This time the compiler says "`void` type not allowed here". The rule is a single one: a `void` call can be written only as a statement on its own, with a semicolon at the end. It cannot be part of an expression.

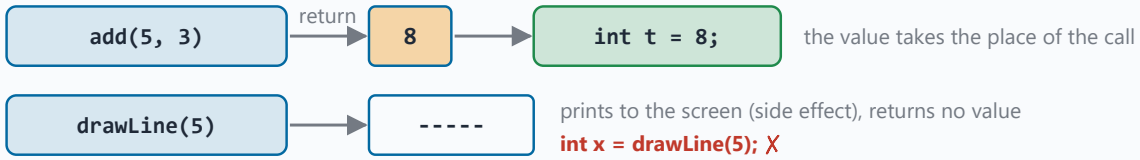


Figure 9.7: A method that returns a value and one that does not. The 8 returned by `add(5, 3)` takes the place of the call and is used in the assignment. `drawLine(5)` only prints to the screen; its result cannot be assigned to a variable.

In a `void` method `return` is not required. The method ends by itself when it reaches the closing curly brace of its body. Even so, a valueless `return;` statement can be written in a `void` method. This statement sends no value; it only ends the method at that point. The early-return pattern is based on this statement.

Common mistake

Thinking that a result must be both printed and returned, or mixing the two up. `System.out.println(x)` prints the value to the screen but sends nothing to the caller. `return x;` sends the value to the caller but prints nothing to the screen. Methods that compute should return the result and leave the decision to print to the caller.

9.5.3 Early return

For some inputs the method has nothing to do. This case is caught at the start of the method and the method is left at once. This way the real work is written in a flat sequence, not buried inside nested `if` blocks.

Definition 9.6

Early return is catching a special case with a condition at the start of the method and ending the method with `return` before it reaches the end of the body. In a `void` method a valueless `return;` is used, and in a method that returns a value, `return` with a value suitable for that case.

```
static void printPositive(int num) {
    if (num <= 0) return;
    System.out.println(num);
}
```

`printPositive` prints only positive numbers. If the number is 0 or negative, `return;` ends the method at once and `println` is never reached. Figure 9.8 shows these two paths with a flow diagram.

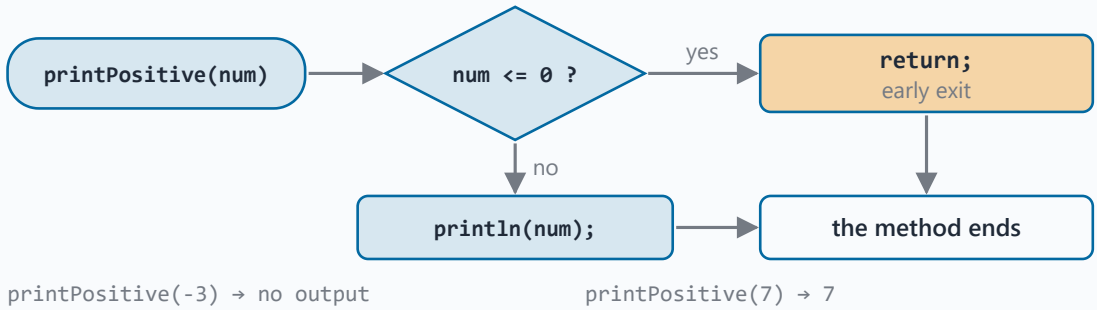


Figure 9.8: The flow of `printPositive`. If `num <= 0`, the Yes branch leaves the method early with `return;`; otherwise the number is printed and the method reaches its end. `printPositive(-3)` prints nothing and `printPositive(7)` prints 7.

In the following program the method is called twice: first with −3, then with 7. Line numbers: 1–4 are the method, and 6 and 7 are the two calls in `main`. Table 9.5 traces the two calls one after the other.

```
static void printPositive(int num) {
    if (num <= 0) return;
    System.out.println(num);
}
// in main:
printPositive(-3);
printPositive(7);
```

Step	Line	Call stack (top first)	Variables (top frame)	Output
1	6	`main()`	—	—
2	1	`printPositive(-3) ← main()`	num=−3	—
3	2	`printPositive(-3) ← main()`	num=−3	—
4	7	`main()`	—	—
5	1	`printPositive(7) ← main()`	num=7	—
6	2	`printPositive(7) ← main()`	num=7	—
7	3	`printPositive(7) ← main()`	num=7	7
8	4	`printPositive(7) ← main()`	num=7	7

Table 9.5: The trace table of the calls `printPositive(-3)` and `printPositive(7)`.

In the first call a frame is opened and num = −3. On line 2, *−3 <= 0* is true and `return;` runs. The method ends at once; the `println` on line 3 is never reached and the output stays empty. `main` continues from the next line, the second call. The second call opens a brand-new frame: num = 7. The num variable of the previous call was deleted long ago; the two calls have no variable in common. This time *7 <= 0* is false, there is no early return and 7 is printed. In the last step execution reaches the closing curly brace on line 4; the `void` method returns by itself here. If a call `printPositive(0)` is added to the full program, the output does not change, because 0 is also caught by the early return:

```
public class PrintPositive {
    static void printPositive(int num) {
        if (num <= 0) return;
        System.out.println(num);
    }

    public static void main(String[] args) {
        printPositive(-3);
        printPositive(7);
        printPositive(0);
    }
}
```

```
7
```

Without early return, the same method would be written as `if (num > 0) { System.out.println(num); }`. For a one-line job there is little difference between the two forms. But if the real work takes ten lines and there are three different special cases, the form without early return gives three levels of nested `if`. With early return, each special case is filtered out in one line at the start of the method, and the real work stays unindented.

Early return is also often used in methods that return a value. In that case the right value for that special case is written next to `return`. The line `if (rest == 0) return 1;` in the digit-count method and the line `if (n < 2) return false;` in the prime method are early returns.

9.5.4 return and type errors

The compiler checks that a method that returns a value really does return a value in every case. In the first example below the method returns a value only when n is positive:

```
public class MissingReturn {
    static int check(int n) {
        if (n > 0) {
            return 1;
        }
    }
}
```

```
MissingReturn.java:6: error: missing return statement
    }
    ^
1 error
```

The message says “missing `return` statement” and points to the closing curly brace of the method. The compiler thinks like this: if n is not positive, the `if` is not entered and execution reaches the end of the method. On this path there is no value to return. In a method that is not `void`, *every path* must end with a `return`. The fix is to close the missing path: add `return 0;` at the end of the method, or write an `else` branch for the `if` and return a value there too.

The compiler does this check without looking at the value of the condition. Even if `if (n > 0) return 1; if (n <= 0) return 0;` is written, the result does not change. For a human the two conditions cover all cases, but the compiler sees only two separate `if`s and gives the same error. The way to convince the compiler is to use `if-else` or to put an unconditional `return` at the end of the method.

The second kind of error is a returned value that does not fit the return type:

```
public class LossyConversion {
    static int f() {
        return 2.5;
    }
}
```

```
LossyConversion.java:3: error: incompatible types: possible lossy conversion from double
to int
        return 2.5;
               ^
1 error
```

The message says “incompatible types: possible lossy conversion from `double` to `int`”. Java does not perform **narrowing conversions**, which can lose information, by itself. If the programmer really wants the fraction discarded, this is stated with an explicit cast: `return (int) 2.5;` compiles and returns 2. If a fractional result is wanted, the return type is made `double`.

There are two more cases where `return` and the return type do not agree. A `void` method cannot return a value, and a valueless `return;` cannot be written in a method that returns a value:

```
public class VoidReturn {
    static void g() {
        return 1;
    }
}
```

```
VoidReturn.java:3: error: incompatible types: unexpected return value
        return 1;
               ^
1 error
```

```
public class NoReturnValue {
    static int h() {
        return;
    }
}
```

```
NoReturnValue.java:3: error: incompatible types: missing return value
        return;
        ^
1 error
```

Situation	Message	Fix
No `return` on one path	`missing return statement`	add a `return` to the missing branch or at the end of the method
`return 2.5;` in an `int` method	`incompatible types: possible lossy conversion from double to int`	an explicit cast with `(int)`, or make the return type `double`

Situation	Message	Fix
`return 1;` in a `void` method	`incompatible types: unexpected return value`	remove the value or change the return type
`return;` in an `int` method	`incompatible types: missing return value`	write a value after `return`
Assigning a `void` call	`incompatible types: void cannot be converted to int`	rewrite the method so that it returns a value
Using a `void` call as an argument	`'void' type not allowed here`	write the call as a separate statement

Table 9.6: Compile errors related to `return` (JDK 21 messages) and their fixes.

Example 9.1

A method `sign` that returns the sign of a number as −1, 0 or 1 has been written like this: `static int sign(int x) { if (x > 0) return 1; else if (x < 0) return -1; }`. Does it compile?

It does not. For x = 0 both conditions are false and execution reaches the end of the method without a `return`. The compiler gives the error `missing return statement`. Once `else return 0;` is added at the end of the chain, or `return 0;` at the end of the method, every path ends with a value: sign(5) = 1, sign(−3) = −1, sign(0) = 0.

9.6 Passing parameters

In a call the value of the argument passes to the parameter. But if the method changes its parameter, does the caller's variable change too? The answer to this question is definite in Java, and it surprises many new programmers.

9.6.1 Pass by value

Definition 9.7

Pass by value means that in a call the value of the argument is copied and written into the parameter. The parameter is a separate box of memory from the argument. Whatever the method does to its parameter, the caller's variable is not affected. Java always passes arguments by value.

For primitive types such as `int`, `double`, `boolean` and `char`, what is passed is the number or the character itself. The caller's variable and the method's parameter are two separate boxes; at the moment of the call, a copy of what is in the first is written into the second. After the copying, the two

boxes are independent of each other.

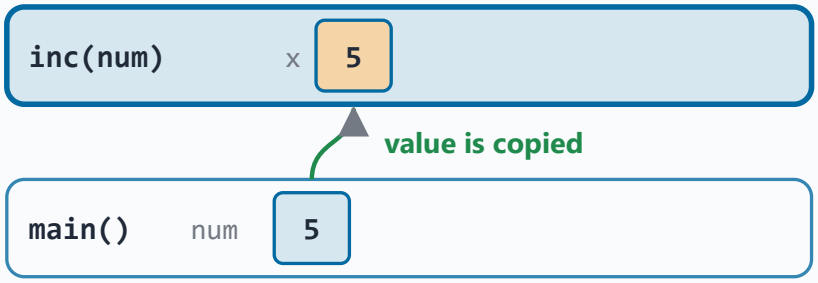


Figure 9.9: In the call `inc(num)`, the value (5) of the num box in `main` is copied into the x box in the frame of `inc`. There are two separate boxes of memory.

For objects, for example arrays, the situation looks different at first sight. An array variable does not contain the array itself but a **reference** that shows where the array is in memory. When an array is sent to a method, it is this reference that is copied. Because the copied reference points to the same array, the method can change the elements of the array. But this is also pass by value: the value copied is a reference. If the method assigns a new array to its parameter, the caller's variable still points to the old array.

9.6.2 inc(num) step by step

The first example is a `void` method that tries to increase a number:

```
static void inc(int x) {
    x++;
}
// in main:
int num = 5;
inc(num);
System.out.println(num);   // 5
```

Line numbers: 1–3 are `inc` and 5–7 are the lines in `main`. When the program is run, it prints 5, not 6:

```
5
```

Step	Line	Call stack (top first)	Variables (top frame)	Output
1	5	`main()`	num=5	—
2	6	`main()`	num=5	—
3	1	`inc(5) ← main()`	x=5	—

Step	Line	Call stack (top first)	Variables (top frame)	Output
4	2	`inc(5) ← main()`	x=6	—
5	3	`inc(5) ← main()`	x=6	—
6	7	`main()`	num=5	5

Table 9.7: The trace table of the call `inc(num)`. x changes; num does not.

In step 1, num = 5 in the frame of `main`. In step 2, `inc(num)` is called. What is sent is not the variable num but its value, 5. In step 3 a separate box is opened in the new frame: x = 5. This is a copy. In step 4, `x++` runs and x = 6. Only the box of `inc` has changed; num in `main` is still 5. In step 5 the method ends, and the frame and the x (6) inside it are deleted. Because `inc` is a `void` method, the value 6 is sent nowhere and is lost completely. In step 6 the value of num, 5, is printed.

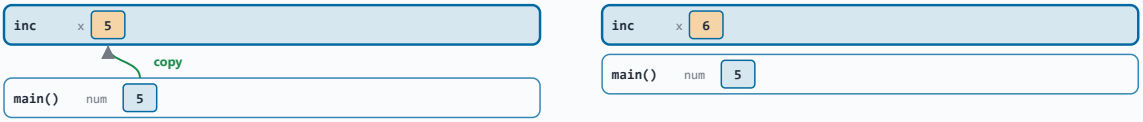


Figure 9.10: (a) The moment of the call: x = 5 in the frame of `inc`, a copy of num. (b) After `x++`: x = 6, but num in `main` is still 5.

9.6.3 The swap trap

Exchanging the values of two variables, that is, swapping them, is a frequent job in programming. A swap is done in three steps with a temporary variable. Putting these three steps into a method looks reasonable:

```
static void swap(int a, int b) {
    int t = a;
    a = b;
    b = t;
}
// in main:
int x = 3, y = 5;
swap(x, y);
System.out.println(x + " " + y);   // 3 5
```

As the comment line shows, the program prints `3 5`, not `5 3`. The swap inside the method is correct, but what is swapped is not x and y themselves but their copies. Table 9.8 shows the situation step by step. Line numbers: 1–5 are `swap` and 7–9 are the lines in `main`.

Step	Line	Call stack (top first)	Variables (top frame)	Output
1	7	`main()`	x=3, y=5	—

Step	Line	Call stack (top first)	Variables (top frame)	Output
2	8	`main()`	x=3, y=5	—
3	1	`swap(3, 5) ← main()`	a=3, b=5	—
4	2	`swap(3, 5) ← main()`	a=3, b=5, t=3	—
5	3	`swap(3, 5) ← main()`	a=5, b=5, t=3	—
6	4	`swap(3, 5) ← main()`	a=5, b=3, t=3	—
7	9	`main()`	x=3, y=5	3 5

Table 9.8: The trace table of the call `swap(x, y)`. a and b are swapped; x and y do not change.

At the moment of the call a = 3 and b = 5; these are copies of x and y and have no connection with them. `int t = a;` puts 3 into the temporary box. With `a = b;`, a = 5; the old value of a is now kept only in t. With `b = t;`, b = 3. At this point the boxes of `swap` have really swapped: a = 5, b = 3. But when the method ends, a, b and t are all deleted. The x and y in `main` have not been touched at all: x = 3, y = 5.

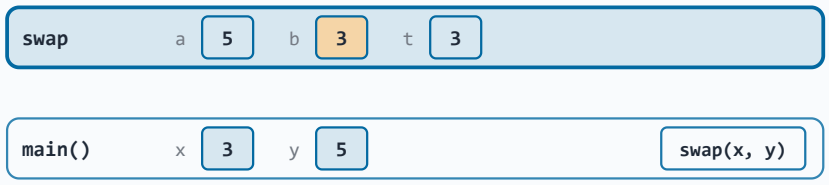


Figure 9.11: After the last line of `swap`: in the frame of `swap`, a = 5, b = 3, t = 3; in the `main` frame below, x = 3, y = 5. The swap has happened only in the copies.

This can also be seen by adding a few print lines inside the method. The version below prints its own variables at the start and the end of the method, and the variables of `main` after the return:

```
public class SwapTrace {
    static void swap(int a, int b) {
        System.out.println("swap start: a=" + a + " b=" + b);
        int t = a;
        a = b;
        b = t;
        System.out.println("swap end: a=" + a + " b=" + b + " t=" + t);
    }

    public static void main(String[] args) {
        int x = 3, y = 5;
        swap(x, y);
        System.out.println("main: x=" + x + " y=" + y);
    }
}
```

```
swap start: a=3 b=5
swap end: a=5 b=3 t=3
main: x=3 y=5
```

The second line of the output shows that the swap succeeded inside the method, and the third line shows that it had no effect outside. In Java it is impossible to write a method that swaps two primitive variables. The swap is either done in the calling code, with a temporary variable, or the values are kept in an array and the elements of the array are swapped.

Common mistake

Thinking that because the parameter and the argument have the same name, the method will change the caller's variable. Even if `static void inc(int num) { num++; }` is written and `inc(num);` is called in `main`, the result does not change. Even if the parameter's name is `num`, it is a separate box in the frame of `inc`. Having the same name does not tie the two variables together.

9.6.4 Passing the result back to the caller

If a method wants to change a variable of the caller, it returns the new value, and the caller assigns this value to its own variable. The corrected version is:

```
public class IncReturn {
    static int inc(int x) {
        return x + 1;
    }

    public static void main(String[] args) {
        int num = 5;
        num = inc(num);
        System.out.println(num);   // 6
    }
}
```

```
6
```

Pass by value applies here too: x is a copy of num, and it is not the method that changes num in `main`. The change is made by `main` itself, with the assignment `num = inc(num);`. This form shows clearly what is happening: the reader can tell from the assignment that num changes on this line. That a

method cannot secretly change the caller's variables may look like a restriction, but it is a guarantee that makes programs understandable. That a call such as `calc(x)` will not spoil x is known without looking at the body of the method.

9.7 Overloading

The area of a square is computed with a single side, the area of a rectangle with two. The two jobs could be given separate names such as `squareArea` and `rectangleArea`. But both really compute "area". Java allows methods that do the same job with different inputs to share the same name.

9.7.1 Same name, different parameter list

Definition 9.8

Method overloading is defining several methods with the same name in the same class. The parameter lists of these methods must differ: in the number of parameters, the parameter types, or the order of the types. In short, their signatures must differ. The compiler binds a call to one of these methods by looking at the number and types of the arguments.

```
static int area(int side) {
    return side * side;
}
static int area(int w, int h) {
    return w * h;
}
// ERROR: only the return type differs
static double area(int side) { … }
```

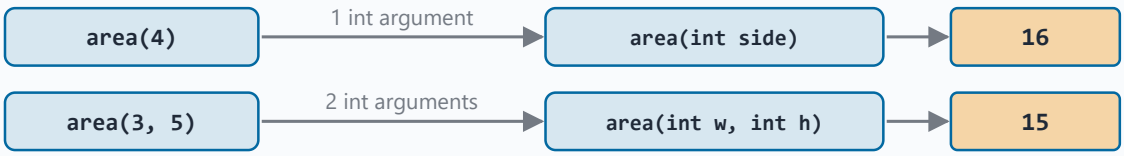
The first two methods are valid overloading. `area(int side)` takes one parameter and `area(int w, int h)` takes two; their signatures are `area(int)` and `area(int, int)`. The third method is wrong: its parameter list is the same as the first one's, and only the return type differs.

```
public class Area {
    static int area(int side) {
        return side * side;
    }

    static int area(int w, int h) {
        return w * h;
    }

    public static void main(String[] args) {
        System.out.println(area(4));
        System.out.println(area(3, 5));
    }
}
```

```
16
15
```



The compiler looks at the number and types of the arguments; it does not look at the return type.

Figure 9.12: In `area(4)` there is a single `int` argument and the one-parameter version is chosen (16). In `area(3, 5)` there are two `int` arguments and the two-parameter version is chosen (15). The compiler looks at the number and types of the arguments; it does not look at the return type.

The most familiar example of overloading is `System.out.println`. The calls `println(5)`, `println(2.5)`, `println("Hello")` and `println(true)` each go to a separate method: `println(int)`, `println(double)`, `println(String)` and `println(boolean)`. The programmer does not have to memorize a separate name for each type. `Math.abs` and `Math.max` are also overloaded for `int`, `long`, `float` and `double`.

9.7.2 Which method does the compiler choose?

The choice among overloaded methods is made at compile time, before the program runs. The compiler knows the type of every argument in the call and looks for the signature that fits these types. The program below shows this decision for two calls:

```
static int area(int side) {
    return side * side;
}
static int area(int w, int h) {
    return w * h;
}
System.out.println(area(4));      // main
System.out.println(area(3, 5));   // main
```

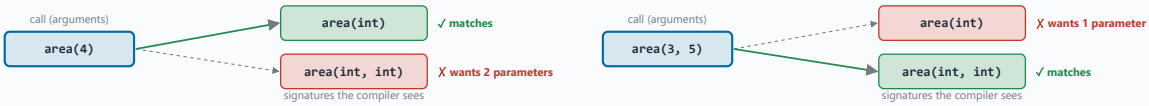


Figure 9.13: The compiler's choice. (a) `area(4)` : `area(int)` matches; `area(int, int)` wants two parameters. (b) `area(3, 5)` : `area(int)` wants one parameter; `area(int, int)` matches.

In the call `area(4)` there is a single `int` argument. `area(int)` fits exactly in number and type; `area(int, int)` is eliminated because it wants two arguments. At run time `area(int)` is called, side = 4 and *4 × 4 = 16* is returned. In the call `area(3, 5)` there are two `int` arguments, and this time only `area(int, int)` fits: w = 3, h = 5 and *3 × 5 = 15* is returned.

When the type of an argument fits no parameter exactly, the compiler tries **widening conversions**. A type can be converted to a wider type without losing information: `char` → `int` → `long` → `float` → `double`. The compiler first looks for an exactly matching signature. If it finds none, it widens the argument, looks at the signatures that fit, and chooses the one that needs the closest widening. Narrowing, on the other hand, is never done automatically. In the program below a `double` version has been added to `area`:

```
public class AreaDouble {
    static int area(int side) {
        return side * side;
    }

    static int area(int w, int h) {
        return w * h;
    }

    static double area(double side) {
        return side * side;
    }

    public static void main(String[] args) {
        System.out.println(area(4));      // area(int)
        System.out.println(area(2.5));    // area(double)
        System.out.println(area(3, 5));   // area(int, int)
        System.out.println(area(4L));     // long → double: area(double)
        System.out.println(area('B'));    // char → int: area(int)
    }
}
```

```
16
6.25
15
16.0
4356
```

Call	Argument types	Method chosen	Why?	Result
`area(4)`	`int`	`area(int)`	exact match	`16`
`area(2.5)`	`double`	`area(double)`	exact match	`6.25`
`area(3, 5)`	`int, int`	`area(int, int)`	exact match	`15`
`area(4L)`	`long`	`area(double)`	`long` cannot be narrowed to `int`; it widens to `double`	`16.0`
`area('B')`	`char`	`area(int)`	`char` → `int` is the closest widening; 'B' = 66	`4356`

Table 9.9: The compiler's choice for the calls in the `AreaDouble` program.

The last two lines need attention. `4L` is a constant of type `long`. It cannot be given to `area(int)`, because going from `long` to `int` is a narrowing. Since `long` can widen to `double`, `area(double)` is chosen and the result is printed as `16.0`, with a decimal point. `'B'` is a `char`. It can widen to both `int` and `double`; the compiler chooses the closer one, `int`. The code of the character 'B' is 66, and *66 × 66 = 4356* is printed. The choice is made entirely at compile time; the value of the argument is not looked at while the program runs.

9.7.3 Faulty overloading

The third method above, `static double area(int side)`, has the same signature as the first one: `area(int)`. Only the return type differs. The compiler does not accept this:

```
public class AreaDuplicate {
    static int area(int side) {
        return side * side;
    }
    static double area(int side) {
        return side * side;
    }
}
```

```
AreaDuplicate.java:5: error: method area(int) is already defined in class AreaDuplicate
    static double area(int side) {
                  ^
1 error
```

The message says "the method `area(int)` is already defined in this class". The reason the return type is not part of the signature is clearly visible here. A compiler meeting a call such as `area(4);` whose result is not used would not know which version was meant. The situation is the same in the call `System.out.println(area(4));`, because `println` accepts both `int` and `double`. The form of a call consists only of the name and the arguments, and the choice can be based only on these.

If there is no matching signature at all, compilation fails again. The call `area(2.5)` does not compile when only `area(int)` and `area(int, int)` are defined:

```
public class AreaError {
    static int area(int side) {
        return side * side;
    }
    static int area(int w, int h) {
        return w * h;
    }
    public static void main(String[] args) {
        System.out.println(area(2.5));
    }
}
```

```
AreaError.java:9: error: incompatible types: possible lossy conversion from double to int
        System.out.println(area(2.5));
                           ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full
output
1 error
```

In this case JDK 21 gives a shortened message. Because the only one-parameter candidate is `area(int)`, it reports the problem directly as a narrowing from `double` to `int`. The note below the message says that the detailed output can be obtained with the `-Xdiags:verbose` option. With this option the compiler says “no suitable method found for area(double)” and lists separately why each of the two candidates does not fit: a lossy conversion in one, a difference in the number of arguments in the other.

A third kind of error is ambiguity. If two different signatures fit equally well through widening, the compiler cannot make a choice:

```java
public class AmbiguousCall {
    static void f(int a, double b) { }
    static void f(double a, int b) { }
    public static void main(String[] args) {
        f(1, 2);
    }
}
```

```
AmbiguousCall.java:5: error: reference to f is ambiguous
        f(1, 2);
        ^
  both method f(int,double) in AmbiguousCall and method f(double,int) in AmbiguousCall
match
1 error
```

In the call `f(1, 2)` both arguments are `int`. For `f(int, double)` the second argument must be widened, and for `f(double, int)` the first. Both options need one widening and neither is better than the other. The compiler says “reference to `f` is ambiguous” and shows both candidates. If one of the arguments is written explicitly as a `double`, the ambiguity disappears:

```
public class Ambiguous {
    static void f(int a, double b) {
        System.out.println("f(int, double)");
    }

    static void f(double a, int b) {
        System.out.println("f(double, int)");
    }

    public static void main(String[] args) {
        f(1, 2.0);
        f(1.0, 2);
    }
}
```

```
f(int, double)
f(double, int)
```

9.8 Variable-length arguments

A method that adds two numbers, another that adds three numbers, a third that adds four... With overloading this road goes on forever. Java offers a syntax that lets a method take as many arguments as wanted.

9.8.1 The int... syntax and its rules

Definition 9.9

Variable-length arguments (**varargs**) are a parameter declared by writing three dots after the parameter type, which accepts zero or more arguments: `int... s`. The compiler packs the values in the call into an array; inside the method `s` is an `int[]` array.

```
static int add(int... s) {
    int t = 0;
    for (int v : s) t += v;
    return t;
}
add(1, 2, 3)   // 6
add()          // 0
add(new int[]{4, 5})   // 9
```

Inside the method, `s` is used like an ordinary array. `s.length` gives how many arguments arrived. The **enhanced for** loop `for (int v : s)` puts the elements of the array into v in turn; one element is processed per iteration. The complete program and its output:

```
public class Varargs {
    static int add(int... s) {
        int t = 0;
        for (int v : s) t += v;
        return t;
    }

    public static void main(String[] args) {
        System.out.println(add(1, 2, 3));
        System.out.println(add());
        System.out.println(add(new int[]{4, 5}));
        System.out.println(add(10));
    }
}
```

```
6
0
9
10
```

For `add(1, 2, 3)` the compiler itself creates the array `new int[]{1, 2, 3}` and passes it to the method. In the call `add()` there are no arguments; this time the compiler passes an empty array, and the result is 0. In the third call the array is given directly: a varargs parameter also accepts an array, and the result is *4 + 5 = 9*. `add(10)` with a single argument is also valid and returns 10.

A varargs parameter has two rules. It must be at the very end of the parameter list, because that is the only way the compiler can tell which of the values in the call go into the array. And there can be only one in a method. Both rules are enforced with the same compiler message:

```
public class VarargsLast {
    static int add(int... s, int x) {
        return x;
    }
}
```

```
VarargsLast.java:2: error: varargs parameter must be the last parameter
    static int add(int... s, int x) {
                           ^
1 error
```

The same message appears when two varargs parameters are written (`int... s, int... t`), because the first is not at the end. Ordinary parameters may come before a varargs parameter. The method below takes a title first and then as many grades as wanted:

```
public class Label {
    static void show(String title, int... grades) {
        System.out.print(title + ":");
        for (int g : grades) System.out.print(" " + g);
        System.out.println(" (count: " + grades.length + ")");
    }

    public static void main(String[] args) {
        show("Ali", 70, 85, 90);
        show("Ayşe", 100);
        show("Can");
        System.out.printf("%d + %d = %d%n", 2, 3, 2 + 3);
    }
}
```

```
Ali: 70 85 90 (count: 3)
Ayşe: 100 (count: 1)
Can: (count: 0)
2 + 3 = 5
```

No grades were given for “Can”; the array is empty, the loop does not run and `grades.length` is 0. The `printf` on the last line of the program is a method in the Java library that uses varargs: its signature is `printf(String, Object...)`. That is why it can take as many values as wanted after the format string.

9.8.2 add(1, 2, 3) step by step

In the `main` below, `add` is called twice:

```
static int add(int... s) {
    int t = 0;
    for (int v : s) t += v;
    return t;
}
// in main:
System.out.println(add(1, 2, 3));
System.out.println(add());
```

The trace of the first call was obtained by running a version in which lines printing the variables on each iteration were added to the method. Table 9.10 was built from this output.

Iteration	v	Operation	t
entry	—	`s = {1, 2, 3}`, `s.length = 3`	0
Iteration 1	1	0 + 1 = 1	1
Iteration 2	2	1 + 2 = 3	3
Iteration 3	3	3 + 3 = 6	6

Table 9.10: The iterations of the enhanced for loop in the call `add(1, 2, 3)`.

At the moment of the call, the array prepared by the compiler is passed to the parameter s: s = {1, 2, 3} and s.length = 3. `int t = 0;` opens a local box for the sum. The enhanced for sets v = 1 in iteration 1 and *t = 0 + 1 = 1*. In iteration 2, v = 2 and *t = 1 + 2 = 3*; in iteration 3, v = 3 and *t = 3 + 3 = 6*. Since there are no more elements in the array, the loop ends. `return t;` returns 6. Because the variable v was declared inside the loop, it has gone out of scope with the loop. The frame of `add` (s and t) is deleted and `main` prints 6.

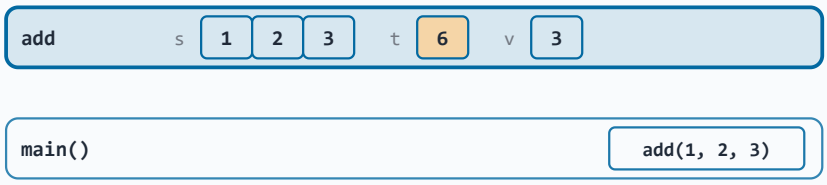


Figure 9.14: The end of iteration 3: in the frame of `add`, the array s is {1, 2, 3}, t = 6 and v = 3.

9.8.3 The empty call

The second call is `add()` and contains no arguments. The compiler passes not `null` but an empty array (`new int[0]`); s.length = 0. `t = 0` is assigned. Because there are no elements in the array, the body of the enhanced for does not run even once and t stays 0. The method returns 0 and the program prints 0.

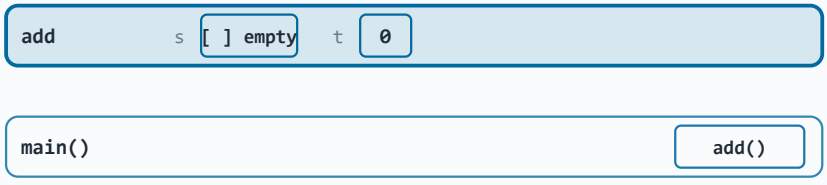


Figure 9.15: In the call `add()`, s is an empty array. The loop body never runs and t stays 0.

Note that the empty call is not an error and that the result is 0. 0 is the **identity element** of addition: adding 0 to a number does not change it. This is why a sum of no numbers being 0 is natural in mathematics too. By the same logic, in a varargs method that computes a product, the accumulator starts at 1 and the empty call returns 1. By contrast, computations such as “largest element” or

“average” have no meaningful answer for empty input. In such methods the empty array must be handled separately at the start. For an integer average, for example, the division *0 / 0* causes an `ArithmeticException` while the program runs.

9.9 Scope

A variable cannot be used from everywhere it is declared. Java gives each variable the right to be visible in a particular region of the program. The boundaries of this region are drawn by curly braces.

9.9.1 Local variables and scope

Definition 9.10

Scope is the region of the program in which a variable can be used by its name. Variables declared inside a method, and the parameters of the method, are called **local variables**. The scope of a local variable starts at the line where it is declared and ends at the closing curly brace of the block in which it is declared. The scope of parameters is the whole body of the method.

Scope has four rules:

- Parameters and variables declared inside a method are local.
- A variable declared in a block, that is, between a pair of curly braces `{ }`, is visible only in that block and in the blocks inside it.
- Two local variables with the same name cannot be declared in the same scope.
- A method cannot reach a local variable of another method.

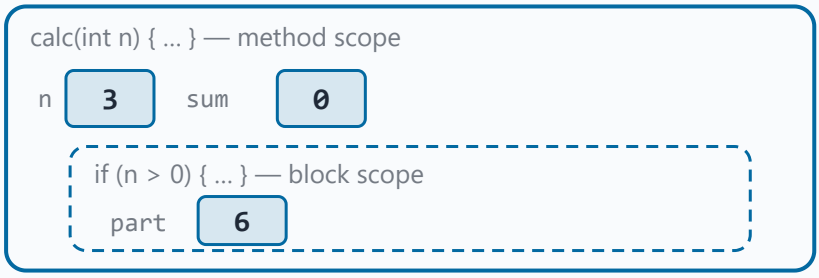


Figure 9.16: Nested scopes. The outer box is the method scope of `calc`: n and sum live here. The dashed inner box is the scope of the `if` block: part exists only here.

The last rule is understood on its own when thought about together with the call stack. The local variables of a method sit in that method's frame. Another method, either when that frame does not yet exist or while it is running inside another frame, cannot reach that variable. The compiler rejects this access from the start:

```
public class OtherMethod {
    static void first() {
        int hidden = 42;
    }
    static void second() {
        System.out.println(hidden);
    }
}
```

```
OtherMethod.java:6: error: cannot find symbol
        System.out.println(hidden);
                           ^
  symbol:   variable hidden
  location: class OtherMethod
1 error
```

The message says “cannot find symbol” and states that the symbol sought is a variable named `hidden`. Inside the method `second` there is nothing known by this name. The way to carry the value from `first` to `second` is for `first` to return the value and for the caller to give this value to `second` as an argument.

9.9.2 calc(3) step by step

The `calc` method contains two nested scopes: the method's own scope and the scope of the `if` block.

```
static int calc(int n) {
    int sum = 0;
    if (n > 0) {
        int part = n * 2;
        sum += part;
    }
    return sum;
}
System.out.println(calc(3));  // main
```

Step	Line	Call stack (top first)	Variables (top frame)	Output
1	9	`main()`	—	—

Step	Line	Call stack (top first)	Variables (top frame)	Output
2	1	`calc(3) ← main()`	n=3	—
3	2	`calc(3) ← main()`	n=3, sum=0	—
4	3	`calc(3) ← main()`	n=3, sum=0	—
5	4	`calc(3) ← main()`	n=3, sum=0, part=6	—
6	5	`calc(3) ← main()`	n=3, sum=6, part=6	—
7	6	`calc(3) ← main()`	n=3, sum=6	—
8	7	`calc(3) ← main()`	n=3, sum=6	—
9	9	`main()`	—	6

Table 9.11: The trace table of the call `calc(3)`. part exists only in steps 5 and 6.

`main` calls `calc(3)` and the method scope opens: n = 3. On line 2, `sum` is declared. Because it is declared in the outermost block of the method, it is visible to the end of the method. On line 3, *3 > 0* is true and the `if` block is entered; a new inner scope opens here. On line 4, `part` is declared and gets the value *3 × 2 = 6*. This variable exists only inside the `if` block. On line 5, the inner block reaches `sum` in the outer scope: *sum = 0 + 6 = 6*. The inner block sees the outer variables; the reverse is not true. On line 6 the block closes and `part` disappears. On line 7, `return sum;` returns 6 and the method scope closes too. The returned 6 becomes the argument to `println` and is printed.

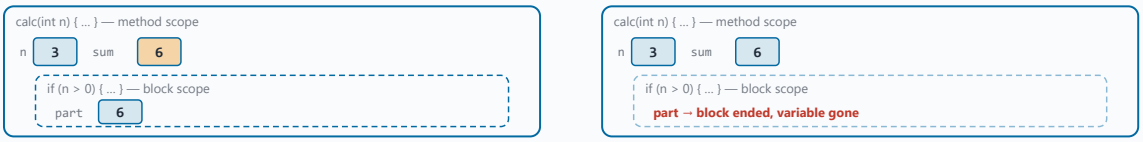


Figure 9.17: (a) After line 5: in the inner block part = 6, in the outer scope sum = 6. (b) After the block closes: part no longer exists; n and sum remain.

Calling the method with a negative number never opens the `if` block: for `calc(-2)`, part is never created and 0 is returned.

```
public class Calc {
    static int calc(int n) {
        int sum = 0;
        if (n > 0) {
            int part = n * 2;
            sum += part;
        }
        return sum;
    }

    public static void main(String[] args) {
        System.out.println(calc(3));
        System.out.println(calc(-2));
    }
}
```

```
6
0
```

Trying to use `part` after the block has closed is a compile error. The version below tries to add part as well on the `return` line:

```
public class PartOutside {
    static int calc(int n) {
        int sum = 0;
        if (n > 0) {
            int part = n * 2;
            sum += part;
        }
        return sum + part;
    }
}
```

```
PartOutside.java:8: error: cannot find symbol
        return sum + part;
                     ^
  symbol:   variable part
  location: class PartOutside
1 error
```

The compiler does not recognize `part` on line 8. This line is outside the `if` block, and there is no variable with this name there. From the compiler's point of view, the situation is no different from the variable never having been declared. If the variable is needed after the block too, its declaration is moved before the block, into the outer scope.

9.9.3 Local variables with the same name

Since an inner block sees the variables of the outer scope, what happens if a new variable with the same name is declared in the inner block? In Java this is an error:

```
public class SameName {
    public static void main(String[] args) {
        int x = 1;
        if (x > 0) {
            int x = 5;   // error
        }
    }
}
```

```
SameName.java:5: error: variable x is already defined in method main(String[])
            int x = 5;   // error
                ^
1 error
```

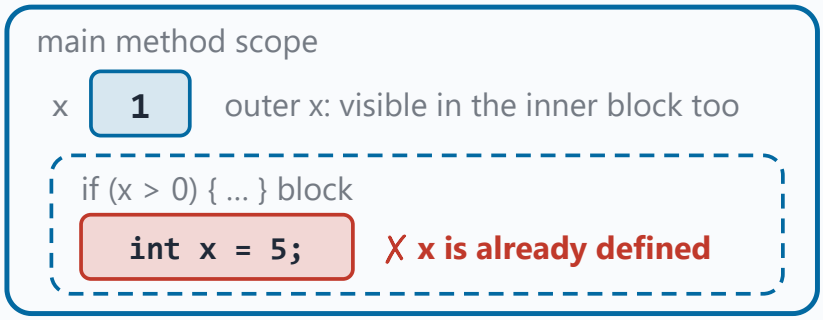


Figure 9.18: The x = 1 in the scope of `main` is visible in the inner `if` block as well. Declaring `int x = 5;` in the inner block is an error: x is already defined.

The message says "variable x is already defined in method `main(String[])`". In some languages a new variable in an inner block temporarily hides the outer one. Java deliberately forbids this among local variables: if two x's were visible at once, the reader could be confused about which x a line refers to.

Parameters are also local variables and live in the outermost scope of the method. So a variable with the same name as a parameter cannot be declared in the method body:

```
public class ParamSameName {
    static int add(int a, int b) {
        int a = 3;
        return a + b;
    }
}
```

```
ParamSameName.java:3: error: variable a is already defined in method add(int,int)
        int a = 3;
            ^
1 error
```

In the message the signature of the method appears with the parameter types: `add(int,int)`. *Assigning* a new value to a parameter (`a = 3;`) is valid; the error comes only from rewriting the type, that is, from redeclaring it.

The same name can be used again after the scope of a variable has ended. When the block closes, the variable inside it is deleted and the name becomes free:

```
public class AfterBlock {
    public static void main(String[] args) {
        for (int i = 0; i < 3; i++) {
            System.out.print(i + " ");
        }
        int i = 7;                 // the scope of the earlier i has ended
        System.out.println(i);
        {
            int k = 1;
            System.out.println(k);
        }
        {
            int k = 2;             // a separate block, a separate k
            System.out.println(k);
        }
    }
}
```

```
0 1 2 7
1
2
```

The scope of the i declared in the `for` header is limited to the loop. When the loop ends, `int i = 7;` can be written. The two k's declared in two separate blocks do not see each other either; each disappears when its own block ends. If the scopes of two variables with the same name do not

overlap, there is no conflict.

9.9.4 Shadowing: a static field and a local variable

Variables are not declared only inside methods. A variable can also be declared inside the class but outside all the methods. Such a variable is called a **field**. A `static` field belongs to the class, is seen by all the methods of the class and lives for the whole program. Unlike local variables, it is not deleted at the end of a call.

Definition 9.11

Shadowing is declaring, inside a method, a local variable with the same name as a field of the class. Within the scope of the local variable, this name refers to the local variable; the field is left in the shade and does not change. The field can still be reached by writing `ClassName.fieldName`.

```
public class FieldShadow {
    static int counter = 100;          // static field

    static void test() {
        int counter = 1;               // the local variable shadows the field
        counter += 5;
        System.out.println("local counter = " + counter);
        System.out.println("field counter = " + FieldShadow.counter);
    }

    public static void main(String[] args) {
        test();
        System.out.println("main: " + counter);
    }
}
```

```
local counter = 6
field counter = 100
main: 100
```

In the method `test`, the moment `int counter = 1;` is written, a local counter is born. From then on, inside the method, the name `counter` refers to this local variable. `counter += 5;` makes the local variable 6. The value of the field stays at 100; the notation `FieldShadow.counter` reaches the field itself and 100 is printed. When `test` ends, the local counter is deleted and the name `counter` in `main` again refers to the field: 100.

Shadowing compiling does not contradict the ban among local variables. The ban prevents two local variables from being visible under the same name at the same time. A field and a local variable, however, are of different kinds, and the field can always be told apart by the class name. Even so, shadowing is often the result of carelessness. The programmer thinks they are changing the field but changes a local copy, and the mistake goes unnoticed for a long time. Giving different names to fields and local variables is the safest way.

9.9.5 Lifetime and initial value of a local variable

Scope says *where* a variable is visible. There is also the question of *when* the variable exists. A local variable is born when its declaration line runs and dies when its block ends. Each time a method is called, its local variables are born anew. No value is remembered from the earlier call. This is why the num in the second call of `printPositive` knows nothing about the −3 of the first call.

A counter that has to carry a value from one call to the next therefore cannot be local; it must be a `static` field. For example, a counter that counts how many times a method has been called, if declared inside the method as `int counter = 0;`, starts from 0 at every call and can never go beyond 1.

There is one more difference between fields and local variables. A `static int` field starts at 0 by itself if no value is given. A local variable has no initial value of its own. The compiler does not allow a local variable that has not been given a value to be read:

```
public class NoInitialValue {
    static int sumTo(int n) {
        int t;
        for (int i = 1; i <= n; i++) t += i;
        return t;
    }
}
```

```
NoInitialValue.java:4: error: variable t might not have been initialized
        for (int i = 1; i <= n; i++) t += i;
                                     ^
NoInitialValue.java:5: error: variable t might not have been initialized
        return t;
               ^
2 errors
```

The message says “variable t might not have been initialized”. The error is reported on two lines: `t += i` reads the old value of t, and `return t` reads it too. The fix is to start the accumulator with `int t = 0;`. The accumulator starts at 0 for addition and at 1 for multiplication.

9.10 Built-in methods: Math

Java's library contains thousands of ready-made methods. One of the most frequently used is the `Math` class. The methods of `Math` are `static`; they are called with the class name without creating an object: `Math.sqrt(16)`. Like the methods written in these notes, they take parameters and return values, and many of them are overloaded.

Call	Result	Return type	Note
`Math.abs(-7)`	`7`	`int`	absolute value; overloaded for `int`, `long`, `float`, `double`
`Math.abs(-7.5)`	`7.5`	`double`	when the argument is `double`, the `double` version is chosen
`Math.max(3, 9)`	`9`	`int`	returns the larger
`Math.max(3, 9.5)`	`9.5`	`double`	3 widens to `double`, and `max(double, double)` is chosen
`Math.min(-4, 2)`	`-4`	`int`	returns the smaller
`Math.sqrt(16)`	`4.0`	`double`	square root; the result is always `double`
`Math.sqrt(-1)`	`NaN`	`double`	a negative number has no real square root: “not a number”
`Math.pow(2, 10)`	`1024.0`	`double`	exponentiation; assigning to an `int` needs `(int)`
`Math.round(2.5)`	`3`	`long`	rounds to the nearest integer
`Math.round(2.4)`	`2`	`long`	rounds down
`Math.round(-2.5)`	`-2`	`long`	exactly at the half, rounds to the larger, that is, to −2

Table 9.12: Frequently used `Math` methods (the results are JDK 21 output).

```
public class MathDemo {
    public static void main(String[] args) {
        System.out.println(Math.abs(-7));
        System.out.println(Math.max(3, 9));
        System.out.println(Math.sqrt(16));
        System.out.println(Math.pow(2, 10));
        System.out.println(Math.round(2.5));
        System.out.println(Math.sqrt(-1));
        System.out.println(Math.abs(-7.5));
        System.out.println(Math.max(3, 9.5));
        System.out.println(Math.round(-2.5));
        System.out.println(Math.round(2.4));
        int p = (int) Math.pow(2, 10);
        System.out.println(p);
        System.out.println(Math.abs(Integer.MIN_VALUE));
        System.out.println(Math.min(-4, 2));
    }
}
```

```
7
9
4.0
1024.0
3
NaN
7.5
9.5
-2
2
1024
-2147483648
-4
```

The decimal point in the numbers `4.0` and `1024.0` in the output comes from the return type being `double`. Even though the result of `Math.sqrt(16)` is a whole number in mathematics, the method always returns `double`. When using a method, the type of the result matters as much as its value. The result of a method whose return type is `double` or `long` cannot be assigned directly to an `int` variable:

```
public class PowInt {
    public static void main(String[] args) {
        int p = Math.pow(2, 10);
    }
}
```

```
PowInt.java:3: error: incompatible types: possible lossy conversion from double to int
        int p = Math.pow(2, 10);
                        ^
1 error
```

```
public class RoundInt {
    public static void main(String[] args) {
        int r = Math.round(2.5);
    }
}
```

```
RoundInt.java:3: error: incompatible types: possible lossy conversion from long to int
        int r = Math.round(2.5);
                          ^
1 error
```

Both errors are narrowing errors. `Math.pow` returns `double`, and `Math.round(double)` returns `long`. If the programmer is sure the result will fit in an `int`, they write an explicit cast: `int p = (int) Math.pow(2, 10);` compiles and gives 1024.

It may be surprising that `Math.round(-2.5)` gives −2, not −3. `Math.round` behaves as if it adds 0.5 to the number and takes the floor of the result: *−2.5 + 0.5 = −2*. Numbers exactly at the half are always rounded toward the larger value, that is, toward positive infinity: 2.5 → 3 and −2.5 → −2. Likewise, the result of the call `Math.abs(Integer.MIN_VALUE)` is negative (−2147483648). 2147483648, the absolute value of the smallest `int` value, is one more than the largest `int` value (2147483647) and does not fit in an `int`. This is why the method that finds the digit count first converts the number to `long`.

Before using a built-in method, read its documentation for the parameters, the return type and the special cases. That `Math.sqrt(-1)` returns `NaN` instead of throwing an error, or that `Math.round` returns `long`, can be learned only by looking at the documentation.

9.11 Method design

A method that compiles and gives the correct result is not always a good method. A hundred-line method that does five jobs at once and both prints its result to the screen and stores it in a field may work, but it is hard to read, test and change. What separates a good method from a bad one is how it is divided and how it is named.

9.11.1 Abstraction

Definition 9.12

Abstraction is hiding the details of a task and offering only a name and an interface that show what the task does. The abstraction of a method is its signature and its **contract**: what it takes, what it returns and under what conditions it works correctly.

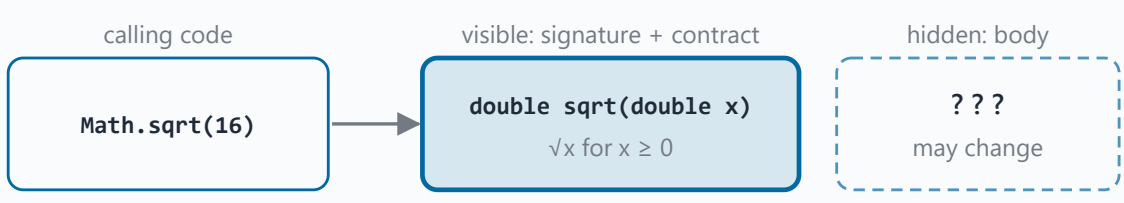


Figure 9.19: Abstraction. The calling code only writes `Math.sqrt(16)`. The visible part is the signature (`double sqrt(double x)`) and the contract (√x for x ≥ 0). The body is hidden and may change.

A method hides the implementation detail from the calling code. A well-named method says *what* is done; *how* it is done stays in the body. A student using `Math.sqrt` does not know whether the square root is computed with Newton's method or with a special processor instruction, and does not need to.

Abstraction has an important benefit: even if the body changes, callers are not affected as long as the signature stays the same. Suppose a prime-number method was first written with a slow loop that tries all divisors from 2 to n − 1. Later it is replaced with a fast version that looks only up to √n. None of the hundred lines that call `isPrime(n)` changes; all of them get faster on their own.

9.11.2 Decomposition

A big task is divided into subtasks, each with a single responsibility. For each subtask the input, the output and what happens on bad input must be determined. The top-level method combines these subtasks in a meaningful order. A top-level method written this way reads like a table of contents. This approach is called **decomposition**.

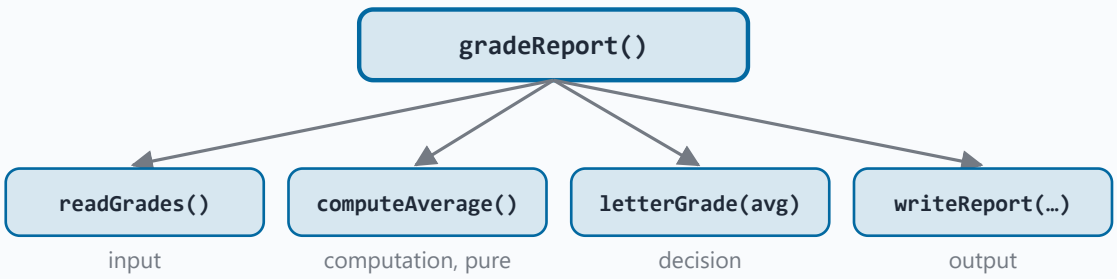


Figure 9.20: Decomposing the grade-report task. `gradeReport()` calls four submethods: one that reads the grades (input), one that computes the average (computation, pure), one that determines the letter grade (decision) and one that writes the report (output).

The following program applies the decomposition in the figure to a small example. Because the reading job is short, it has not been made a separate method; it is written in the first three lines of `gradeReport`. The letter-grade boundaries have been chosen as an example.

```java
import java.util.Scanner;

public class GradeReport {
    static double computeAverage(int a, int b, int c) {
        return (a + b + c) / 3.0;
    }

    static String letterGrade(double avg) {
        if (avg >= 85) return "AA";
        if (avg >= 70) return "BB";
        if (avg >= 60) return "CC";
        if (avg >= 50) return "DD";
        return "FF";
    }

    static void writeReport(double avg, String letter) {
        System.out.println("Average: " + Math.round(avg * 100) / 100.0);
        System.out.println("Letter grade: " + letter);
    }

    static void gradeReport(Scanner input) {
        int g1 = input.nextInt();               // read the grades
        int g2 = input.nextInt();
        int g3 = input.nextInt();
        double avg = computeAverage(g1, g2, g3);
        writeReport(avg, letterGrade(avg));
    }

    public static void main(String[] args) {
        gradeReport(new Scanner(System.in));
    }
}
```

Outputs for three different inputs:

Input	Average (computation)	Output
`70 85 91`	246 / 3 = 82	`Average: 82.0 Letter grade: BB`
`50 60 71`	181 / 3 ≈ 60,33	`Average: 60.33 Letter grade: CC`

Input	Average (computation)	Output
`40 45 52`	137 / 3 ≈ 45,67	`Average: 45.67 Letter grade: FF`

Table 9.13: The output of the `GradeReport` program for three inputs.

Each method has a single job, and each can be tried separately. `computeAverage` only computes. Note that the division is done by `3.0`: if `(a + b + c) / 3` were written, integer division between two `int`s would be done, and for 181 / 3 the result would be 60 instead of 60.33. The return type being `double` does not by itself make this division fractional; the conversion happens after the division is finished. `letterGrade` only makes a decision; because it is written with successive early returns, each threshold is one line. `writeReport` is concerned only with output. To round the average to two decimal places, `Math.round(avg * 100) / 100.0` is used: 60.333… × 100 = 6033.3… → 6033 → 60.33. If the letter-grade boundaries change, only `letterGrade` changes; if the format of the report changes, only `writeReport` changes.

9.11.3 Principles of good method design

Whether a method is well designed can be tested with a few questions:

Principle	Question	Bad example → good example
Single task	Can the method be described in one sentence?	a `process()` that reads, computes and writes → `readGrades`, `computeAverage`, `writeReport`
Meaningful name	Does the name tell the result or the action?	`f(x)`, `op2()` → `digitCount(x)`, `isEven(x)`
Few parameters	Are all the inputs necessary?	an unused `int mod` parameter → remove the parameter
Clear result	Are the return and the side effects clear?	a method that both prints and returns → a method that only returns
Testability	Can the method be tried with independent examples?	a computing method that reads from the keyboard → a method that takes the value as a parameter

Table 9.14: The principles of good method design and the question for each principle.

These principles support one another. It is easy to give a meaningful name to a single-task method. A method with few parameters that returns its result is easy to test. There is no exact limit on length. Even so, a method that exceeds a screen in height is a candidate for splitting. If its body has sections that separate with “now do this”, each of these sections should probably be a separate method.

9.12 Example methods

The examples below bring together the concepts about methods on small, familiar problems. Each method is first written, then executed step by step with the call stack or a trace table.

9.12.1 The contract first

Before a method is written, what it will do must be stated clearly. The name of the method should say what it does, the valid range of the parameters should be defined, and the meaning of the return value and what happens on bad input should be known. The sum of this information is called the method's contract.

Definition 9.13

A **contract** is the agreement between a method and its caller. A **precondition** is a condition the caller must satisfy (e.g. “n ≥ 0”). A **postcondition** is the result the method guarantees when the precondition is satisfied (e.g. “returns the number of digits of n”).

```
// Task  : returns the number of decimal digits.
// Input : any int (may be negative).
// Output: 1..10; 1 for 0, 3 for -507.
static int digitCount(int num) { … }
```

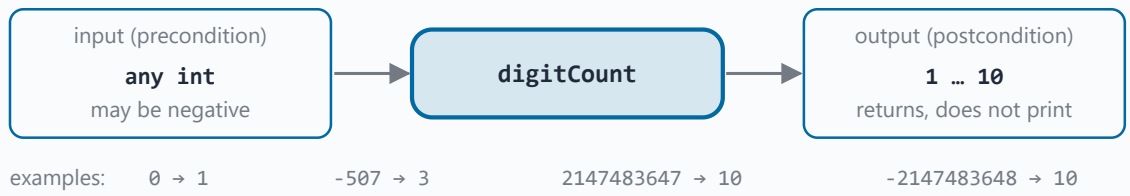


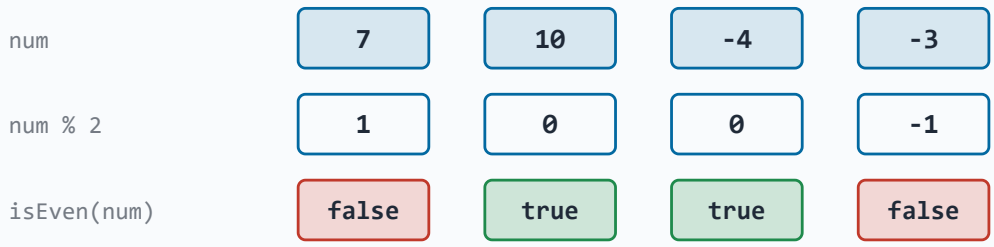
Figure 9.21: The contract of `digitCount`. Precondition: any `int`, possibly negative. Postcondition: returns a number between 1 and 10 and does not print. Examples: 0 → 1, −507 → 3, 2147483647 → 10, −2147483648 → 10.

The examples in the contract are test cases determined before the method is written. They deliberately include boundary values: 0 has one digit but is an input for which the loop will never run; the largest and smallest `int` values are the longest numbers. After the body is written, all these examples are tried. In a method started without a contract, the question “what should be returned for 0?” is often asked only when a bug appears.

9.12.2 Even or odd?

```
static boolean isEven(int num) {
    return num % 2 == 0;
}
```

If the remainder is 0 when a number is divided by 2, the number is even. `isEven` returns the result of this comparison, that is, a `boolean` value, directly. Writing it as `if (num % 2 == 0) return true; else return false;` is also correct, but needlessly long: the expression `num % 2 == 0` is already `true` or `false`.



In Java the remainder of a negative number can be negative (-3 % 2 = -1); that is why the test is == 0.

Figure 9.22: `isEven` for four inputs: 7 % 2 = 1 → `false`; 10 % 2 = 0 → `true`; −4 % 2 = 0 → `true`; −3 % 2 = −1 → `false`.

The last column of the figure shows an important detail. In Java the remainder of a negative number can be negative: the sign of the remainder follows the sign of the dividend, and *−3 % 2 = −1*. Writing the odd-number test as `num % 2 == 1` is therefore wrong; for −3 it gives `false` and treats −3 as even. Testing evenness with `== 0` works correctly for every sign. The real output of the program gives, on each line, the number, the remainder and the result of the method:

```
public class IsEven {
    static boolean isEven(int num) {
        return num % 2 == 0;
    }

    public static void main(String[] args) {
        if (isEven(7)) System.out.println("even");
        else System.out.println("odd");
        int[] samples = {7, 10, -4, -3, 0};
        for (int s : samples)
            System.out.println(s + " " + (s % 2) + " " + isEven(s));
    }
}
```

```
odd
7 1 false
10 0 true
-4 0 true
-3 -1 false
0 0 true
```

The method does not print; it returns the result. That is why it is used directly in the `if` condition. When the call `isEven(7)` is traced, num = 7 in the frame, *7 % 2 = 1* and, since *1 == 0* is false, `false` is returned. The return value takes the place of the call: `if (false)`. The then branch is skipped and the `else` branch prints "odd". It is the caller, not the method, that has decided what to print. The same `isEven` can be used unchanged in another program to count even numbers or to filter a list.

9.12.3 Digit count

```
static int digitCount(int num) {
    long rest = Math.abs((long) num);
    if (rest == 0) return 1;
    int count = 0;
    while (rest > 0) {
        count++;
        rest /= 10;
    }
    return count;
}
System.out.println(digitCount(-507));
```

The method first takes the absolute value of the number, because −507 and 507 have the same number of digits. It then makes the early return for 0. After that, each time the number is divided by 10 it loses one digit; the method counts how many divisions it takes until the number becomes 0. The result is not printed but returned; the caller uses it however it likes.

Table 9.15 gives the iterations of the loop in the call `digitCount(-507)`. Before the loop, *rest = |−507| = 507*; since *507 == 0* is false, there is no early return and the loop is entered with count = 0.

Iteration	Condition `rest > 0`	count	rest
Iteration 1	507 > 0 → true	1	507 / 10 = 50
Iteration 2	50 > 0 → true	2	50 / 10 = 5
Iteration 3	5 > 0 → true	3	5 / 10 = 0
exit	0 > 0 → false	3	—

Table 9.15: The iterations of the `while` loop in the call `digitCount(-507)`. `rest /= 10` is integer division.

In each iteration `rest /= 10` drops the last digit of the number: 507 → 50 → 5 → 0. The loop runs three times and 3 is returned. This call was made first to get the argument of the `println` in `main`; the returned 3 took the place of the call and `println(3)` ran.



Figure 9.23: (a) After `Math.abs`: num = −507, rest = 507. (b) The loop has finished: rest = 0, count = 3; the frame closes with `return 3`.

All the examples in the contract have actually been tried:

```
public class DigitCount {
    static int digitCount(int num) {
        long rest = Math.abs((long) num);
        if (rest == 0) return 1;
        int count = 0;
        while (rest > 0) {
            count++;
            rest /= 10;
        }
        return count;
    }

    public static void main(String[] args) {
        System.out.println(digitCount(-507));
        System.out.println(digitCount(0));
        System.out.println(digitCount(9));
        System.out.println(digitCount(10));
        System.out.println(digitCount(2147483647));
        System.out.println(digitCount(-2147483648));
    }
}
```

```
3
1
1
2
10
10
```

The `(long)` cast in the first line also has a reason. The absolute value of −2147483648, the smallest `int` value, does not fit in an `int`, and `Math.abs` again returns −2147483648 for this number. A version written without the cast therefore gives a wrong result:

```
public class DigitWrong {
    static int digitCount(int num) {
        int rest = Math.abs(num);      // without converting to long
        if (rest == 0) return 1;
        int count = 0;
        while (rest > 0) {
            count++;
            rest /= 10;
        }
        return count;
    }

    public static void main(String[] args) {
        System.out.println(Math.abs(-2147483648));
        System.out.println(digitCount(-2147483648));
        System.out.println(digitCount(-507));
    }
}
```

```
-2147483648
0
3
```

In the faulty version, rest stays negative. Because it is not equal to 0, there is no early return. Because the condition *rest > 0* is false the first time, the loop never runs and 0 is returned. Yet the contract said "between 1 and 10". The correct version first converts the number to `long`. In a `long`, 2147483648 is easily represented, and the `long` version of `Math.abs` gives the right result. This is a bug that shows up only when tried with a boundary value, and it shows the value of writing boundary values into the contract.

9.12.4 Is it prime?

A number greater than 1 that has no divisors other than itself and 1 is called a prime number. To find out whether a number is prime, the possible divisors are tried in order. The moment a divisor is found, the answer is settled and the method makes an early return.

```
static boolean isPrime(int n) {
    if (n < 2) return false;
    for (int i = 2; i <= n / i; i++)
        if (n % i == 0) return false;
    return true;
}
```

The expression `i <= n / i` in the loop condition may look strange at first. This condition makes i be tried up to the square root of n. The reason is this: if n can be written as the product of two numbers, that is, $n = p \times q$, then at least one of p and q cannot be greater than √n. If both were greater than √n, their product would exceed n. So if n has a divisor greater than √n, it also has a partner smaller than √n, and that partner would already have been tried. For example, 91 = 7 × 13 and √91 ≈ 9.54. There is no need to wait for 13; the divisor is found when 7 is tried.

The condition `i <= n / i` is the form of the condition `i * i <= n` written with integer division. This form prevents the product `i * i` from exceeding the `int` limit when n is very large. Table 9.16 shows how the condition works for 37. The values in the table were obtained by running a version of the loop with a print line added, on JDK 21.

i	`n / i`	Condition `i <= n / i`	`n % i`	Does it divide?
2	37 / 2 = 18	2 ≤ 18 → true	37 % 2 = 1	no
3	37 / 3 = 12	3 ≤ 12 → true	37 % 3 = 1	no
4	37 / 4 = 9	4 ≤ 9 → true	37 % 4 = 1	no
5	37 / 5 = 7	5 ≤ 7 → true	37 % 5 = 2	no
6	37 / 6 = 6	6 ≤ 6 → true	37 % 6 = 1	no
7	37 / 7 = 5	7 ≤ 5 → false	—	the loop ends: `return true`

Table 9.16: `isPrime(37)`: only the values of i with i ≤ n / i are tried (√37 ≈ 6.08).

Five values, i = 2 to 6, are tried and none divides 37. At i = 7, *37 / 7 = 5* and *7 ≤ 5* is false; the loop ends and the method returns `true`. A simple version that tries all the numbers from 2 to 36 would compute 35 remainders; this version computes only 5. The result is the same. For a number that is not prime, the loop ends at the first divisor. For 49 = 7 × 7, at i = 7, *7 ≤ 49 / 7 = 7* is true and *49 % 7 = 0* is found:

i	`n / i`	Condition `i <= n / i`	`n % i`	Does it divide?
2	49 / 2 = 24	2 ≤ 24 → true	49 % 2 = 1	no
3	49 / 3 = 16	3 ≤ 16 → true	49 % 3 = 1	no
4	49 / 4 = 12	4 ≤ 12 → true	49 % 4 = 1	no
5	49 / 5 = 9	5 ≤ 9 → true	49 % 5 = 4	no
6	49 / 6 = 8	6 ≤ 8 → true	49 % 6 = 1	no
7	49 / 7 = 7	7 ≤ 7 → true	49 % 7 = 0	yes: `return false`

Table 9.17: `isPrime(49)`: the condition is still true for i = 7, because 7 × 7 = 49.

This example shows why `<=`, not `<`, is used in the condition. If `i < n / i` were written, 7 would never be tried and 49 would be taken for a prime. Perfect squares are good test inputs for catching this kind of boundary error.

The method can be traced for 15 and 7. In `isPrime(15)`, for n = 15, first *15 < 2* is false. For i = 2, *2 ≤ 15 / 2 = 7* is true but *15 % 2 = 1*; i increases. For i = 3, *3 ≤ 15 / 3 = 5* is true and *15 % 3 = 0*: `return false` runs and the remaining iterations never run. In `isPrime(7)`, for i = 2, *2 ≤ 7 / 2 = 3* is true and *7 % 2 = 1*. For i = 3, *3 ≤ 7 / 3 = 2* is false: no divisor has been found up to √7, the loop ends without an early return and `true` is returned.

```
public class IsPrime {
    static boolean isPrime(int n) {
        if (n < 2) return false;
        for (int i = 2; i <= n / i; i++)
            if (n % i == 0) return false;
        return true;
    }

    public static void main(String[] args) {
        System.out.println(isPrime(15));
        System.out.println(isPrime(7));
        System.out.println(isPrime(37));
        System.out.println(isPrime(1) + " " + isPrime(2) + " " + isPrime(4) + " " +
isPrime(49) + " " + isPrime(97));
        for (int k = 1; k <= 30; k++)
            if (isPrime(k)) System.out.print(k + " ");
        System.out.println();
    }
}
```

```
false
true
true
false true false false true
2 3 5 7 11 13 17 19 23 29
```

The last line shows the primes obtained by calling the method in a loop with the numbers from 1 to 30. Note that `isPrime(1)` returns `false` and `isPrime(2)` returns `true`. 1 is not prime and is eliminated by the early return in the first line. For 2, the loop condition *2 ≤ 2 / 2 = 1* is false from the start; the loop never runs and `true` is returned.

9.12.5 GCD: Euclid's algorithm

The greatest common divisor (GCD) of two numbers is the largest number that divides both without a remainder. Instead of trying all the numbers from 1 up to the smaller number, the roughly 2,300-year-old Euclid's algorithm offers a much faster way: *gcd(a, b) = gcd(b, a % b)*, and when b = 0 the answer is a.

```
static int gcd(int a, int b) {
    while (b != 0) {
        int r = a % b;
        a = b;
        b = r;
    }
    return Math.abs(a);
}
System.out.println(gcd(48, 18));  // main
```

The correctness of the algorithm rests on this observation: every number that divides both a and b also divides the difference *a % b = a − k × b*. Conversely, every number that divides both b and a % b also divides a. So the pair (a, b) and the pair (b, a % b) have the same common divisors, and therefore the same GCD. The numbers get smaller in each iteration. Finally b = 0 and *gcd(a, 0) = a*, because every number divides 0.

Iteration	`r = a % b`	a	b	Condition `b != 0`
start	—	48	18	18 != 0 → true
Iteration 1	48 % 18 = 12	18	12	12 != 0 → true
Iteration 2	18 % 12 = 6	12	6	6 != 0 → true
Iteration 3	12 % 6 = 0	6	0	0 != 0 → false

Table 9.18: The iterations of `gcd(48, 18)` (the values are taken from a trace version run on JDK 21).

At the start a = 48, b = 18 and *18 != 0* is true. In iteration 1, *r = 48 % 18 = 12* (48 = 2 × 18 + 12); a = 18, b = 12. This means *gcd(48, 18) = gcd(18, 12)*. In iteration 2, *r = 18 % 12 = 6*; a = 12, b = 6. In iteration 3, *r = 12 % 6 = 0*; a = 6, b = 0. The condition *0 != 0* is false and 6 is returned. Check: *48 = 6 × 8*, *18 = 6 × 3*, and 8 and 3 have no common divisor other than 1. Only three iterations were enough.



Figure 9.24: (a) The frame of `gcd` after iteration 1: a = 18, b = 12, r = 12. (b) The loop has finished: a = 6, b = 0; `return 6`.

If the numbers are given in the reverse order, the first iteration swaps them by itself: since *18 % 48 = 18*, `gcd(18, 48)` reaches the state of `gcd(48, 18)` after the first iteration. The method also works correctly with 0: `gcd(7, 0)` never enters the loop and returns 7. The `Math.abs` in the last line makes the result positive for negative inputs: `gcd(-12, 18)` returns 6, not −6.

```
public class Gcd {
    static int gcd(int a, int b) {
        while (b != 0) {
            int r = a % b;
            a = b;
            b = r;
        }
        return Math.abs(a);
    }

    static int lcm(int a, int b) {
        if (a == 0 || b == 0) return 0;
        return Math.abs(a / gcd(a, b) * b);
    }

    public static void main(String[] args) {
        System.out.println(gcd(48, 18));
        System.out.println(lcm(12, 18));
        System.out.println(gcd(18, 48) + " " + gcd(17, 5) + " " + gcd(7, 0) + " " +
gcd(100, 75));
        System.out.println(lcm(4, 6) + " " + lcm(5, 7) + " " + lcm(8, 12));
    }
}
```

```
6
36
6 1 7 25
12 35 24
```

9.12.6 LCM: calling a method from a method

The least common multiple (LCM) of two numbers is easily found with the GCD: *lcm(a, b) = a × b / gcd(a, b)*. Since the method that computes the GCD has already been written, the `lcm` method calls it; the real computation is a single line:

```
static int lcm(int a, int b) {
    if (a == 0 || b == 0) return 0;
    return Math.abs(a / gcd(a, b) * b);
}
// gcd: Euclid's algorithm
System.out.println(lcm(12, 18));  // main
```

Step	Line	Call stack (top first)	Variables (top frame)	Output
1	6	`main()`	—	—
2	1	`lcm(12, 18)` ← `main()`	a=12, b=18	—

Step	Line	Call stack (top first)	Variables (top frame)	Output
3	3	`gcd(12, 18) ← lcm(12, 18) ← main()`	a=12, b=18	—
4	3	`gcd(12, 18) ← lcm(12, 18) ← main()`	a=6, b=0	—
5	3	`lcm(12, 18) ← main()`	a=12, b=18	—
6	3	`lcm(12, 18) ← main()`	a=12, b=18	—
7	6	`main()`	—	36

Table 9.19: The trace table of the call `lcm(12, 18)`. In steps 3 and 4 there are three frames on the stack.

`main` calls `lcm(12, 18)` and the frame of `lcm` is opened: a = 12, b = 18. Because neither is zero, the early return on line 2 is skipped. To compute the expression in the `return` line, `gcd(a, b)` must be called first. There are three frames on the stack. `gcd` runs its own loop: (12, 18) → (18, 12) → (12, 6) → (6, 0) and returns 6. Although the names are the same, the a and b of `gcd` are boxes separate from the a and b of `lcm`. While the a of `gcd` goes down to 6, the a of `lcm` stays at 12. 6 takes the place of the call and the expression is computed: *12 / 6 × 18 = 2 × 18 = 36*. `lcm` returns 36 and 36 is printed.

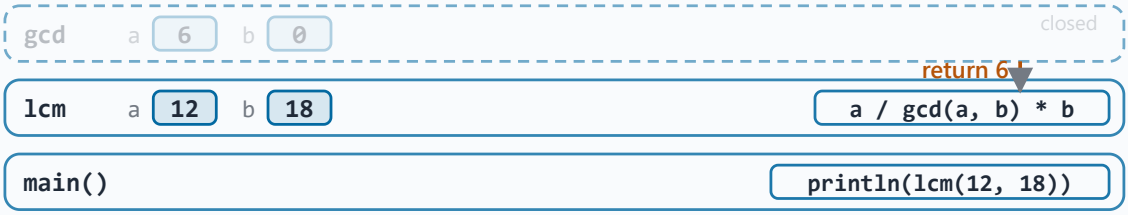


Figure 9.25: `gcd` is above `lcm`: the a = 6, b = 0 of `gcd` is about to return 6; the a = 12, b = 18 of `lcm` below is unchanged.

There is a reason why the formula in the code is written as `a / gcd(a, b) * b` instead of `a * b / gcd(a, b)`. Dividing first keeps the intermediate result from growing. The product a × b can be too big to fit in an `int`. But a is always divisible by the GCD without a remainder, and the quotient is small. The difference between the two forms shows up with large numbers:

```
public class LcmOverflow {
    static int gcd(int a, int b) {
        while (b != 0) {
            int r = a % b;
            a = b;
            b = r;
        }
        return Math.abs(a);
    }

    static int lcmMultiplyFirst(int a, int b) {
        return a * b / gcd(a, b);
    }

    static int lcm(int a, int b) {
        if (a == 0 || b == 0) return 0;
        return Math.abs(a / gcd(a, b) * b);
    }

    public static void main(String[] args) {
        System.out.println(lcmMultiplyFirst(12, 18) + " " + lcm(12, 18));
        System.out.println(lcmMultiplyFirst(100000, 300000) + " " + lcm(100000, 300000));
    }
}
```

```
36 36
-647 300000
```

For 12 and 18 both forms give 36. For 100000 and 300000, however, the product *100000 × 300000 = 30,000,000,000* exceeds the upper limit of `int` (about 2.1 billion). The overflowed product turns into a meaningless negative number and the result comes out as −647. The form that divides first gives the right answer with *100000 / 100000 × 300000 = 300000*. The sneaky thing about this bug is that neither the compiler nor the runtime gives any warning.

The first line of `lcm` also covers an edge case. If one of the numbers is 0, the common multiple is 0; when both numbers are 0, `gcd(0, 0)` returns 0, and without the check the division `0 / 0` would throw an `ArithmeticException`. The outer `Math.abs` makes the result positive for negative inputs: `lcm(-4, 6)` gives 12.

Calling a method from within a method avoids code duplication. The loop that computes the GCD has not been written again inside `lcm`. If a bug is found and fixed in `gcd`, `lcm` is fixed automatically too.

9.12.7 Reuse: max3

To find the largest of three numbers, there is no need to write a new chain of comparisons. The method that gives the larger of two numbers is used twice: first the larger of the first two numbers is found, then this result is compared with the third number.

```
static int max(int a, int b) {
    if (a > b) return a;
    return b;
}
static int max3(int a, int b, int c) {
    return max(max(a, b), c);
}
System.out.println(max3(4, 9, 7));  // main
```

Step	Line	Call stack (top first)	Variables (top frame)	Output
1	8	`main()`	—	—
2	5	`max3(4, 9, 7) ← main()`	a=4, b=9, c=7	—
3	6	`max3(4, 9, 7) ← main()`	a=4, b=9, c=7	—
4	1, 2	`max(4, 9) ← max3(4, 9, 7) ← main()`	a=4, b=9	—
5	3	`max(4, 9) ← max3(4, 9, 7) ← main()`	a=4, b=9	—
6	6	`max3(4, 9, 7) ← main()`	a=4, b=9, c=7	—
7	2	`max(9, 7) ← max3(4, 9, 7) ← main()`	a=9, b=7	—
8	6	`max3(4, 9, 7) ← main()`	a=4, b=9, c=7	—
9	8	`main()`	—	9

Table 9.20: The trace table of the call `max3(4, 9, 7)`. `max` is called twice, with two separate frames.

In the frame of `max3`, a = 4, b = 9, c = 7. The line `return max(max(a, b), c);` has a nested call, and the inner one is computed first: `max(4, 9)`. In the new frame a = 4, b = 9. Since *4 > 9* is false, the first `return` is skipped and `return b;` returns 9. The expression has simplified to `max(9, 7)`. The same method is called a second time, with a brand-new frame: a = 9, b = 7. *9 > 7* is true and `return a;` returns 9. `max3` returns this 9 too, and 9 is printed.



Figure 9.26: (a) The inner call: the frame of `max(4, 9)` is above `max3`. (b) The outer call: `max(9, 7)` returns 9.

In this design the comparison logic is in one place. If there is a bug, for example if `<` was written by mistake instead of `>`, it is fixed in a single place. The order in which the numbers are given does not change the result; it works correctly with equal and negative numbers too:

```
public class Max3 {
    static int max(int a, int b) {
        if (a > b) return a;
        return b;
    }

    static int max3(int a, int b, int c) {
        return max(max(a, b), c);
    }

    public static void main(String[] args) {
        System.out.println(max3(4, 9, 7));
        System.out.println(max3(9, 4, 7) + " " + max3(4, 7, 9) + " " + max3(5, 5, 5) + " "
+ max3(-2, -7, -1));
    }
}
```

```
9
9 9 5 -1
```

9.12.8 Methods that return double: temperature conversion

The conversion between Celsius and Fahrenheit is done with the formulas $F = C \times 9 / 5 + 32$ and $C = (F - 32) \times 5 / 9$. Since the results can be fractional, both methods take `double` and return `double`.

```
static double fahrenheit(double c) {
    return c * 9 / 5 + 32;
}
static double celsius(double f) {
    return (f - 32) * 5 / 9;
}
double f = fahrenheit(25);          // main
System.out.println(f);
System.out.println(celsius(f));
```

Step	Line	Call stack (top first)	Variables (top frame)	Output
1	7	`main()`	—	—
2	1	`fahrenheit(25.0) ← main()`	c=25.0	—
3	2	`fahrenheit(25.0) ← main()`	c=25.0	—
4	7	`main()`	f=77.0	—

Step	Line	Call stack (top first)	Variables (top frame)	Output
5	8	`main()`	f=77.0	77.0
6	9	`celsius(77.0) ← main()`	f=77.0	77.0
7	5	`celsius(77.0) ← main()`	f=77.0	77.0
8	9	`main()`	f=77.0	77.0 / 25.0

Table 9.21: The trace table of the calls `fahrenheit(25)` and `celsius(f)`.

In the call `fahrenheit(25)` the argument is `int` but the parameter is `double`. Widening is done automatically and c = 25.0. In the body, *25.0 × 9 = 225.0*, *225.0 / 5 = 45.0* and *45.0 + 32 = 77.0* are computed. In `main`, f = 77.0, and because it is a `double` it is printed in the form `77.0`, with a decimal point. Then `celsius(f)` is called. The value of f is copied into `celsius`'s own f parameter; again, having the same name creates no link. *(77.0 − 32) × 5 / 9 = 45.0 × 5 / 9 = 225.0 / 9 = 25.0* is found and `25.0` is printed. The two methods are inverses of each other. Giving the result of one to the other and getting back to the starting value is an easy way to test the two together.

```
public class Temperature {
    static double fahrenheit(double c) {
        return c * 9 / 5 + 32;
    }

    static double celsius(double f) {
        return (f - 32) * 5 / 9;
    }

    public static void main(String[] args) {
        double f = fahrenheit(25);
        System.out.println(f);
        System.out.println(celsius(f));
        System.out.println(fahrenheit(100) + " " + fahrenheit(-40) + " " + celsius(98.6));
        System.out.println(5 / 9 * (f - 32));
        System.out.println((f - 32) * 5 / 9);
    }
}
```

```
77.0
25.0
212.0 -40.0 37.0
0.0
25.0
```

The third line checks a few known values: the boiling point of water, 100 °C = 212 °F, −40, where the two scales meet, and human body temperature, 98.6 °F = 37 °C. The fourth line shows a common mistake. If the formula is written as `5 / 9 * (f - 32)`, `5 / 9` is computed first. Integer division

between two `int`s gives 0, and the result is `0.0` whatever the value of f. In the correct form, `(f - 32)`, a `double`, is computed first; the following multiplication and division are done in `double` arithmetic and the right result (`25.0`) is obtained.

9.12.9 Pure methods and side effects

> **Definition 9.14**
>
> A **side effect** is any change a method leaves in the world apart from returning a value: printing to the screen, changing a field, adding a record to a file. A **pure method** is a method that always returns the same result for the same inputs and has no side effects.

Two small methods show the difference clearly:

```
public class PureDemo {
    static int square(int x) {
        return x * x;
    }

    static void printSquare(int x) {
        System.out.println(x * x);
    }

    public static void main(String[] args) {
        int t = square(3) + square(4);
        System.out.println(t);
        printSquare(4);
        System.out.println(square(square(2)));
    }
}
```

```
25
16
16
```

`square` is pure. `square(4)` is always 16 and can be used anywhere: inside a sum (*9 + 16 = 25*), as an argument to another call (`square(square(2))` = *square(4) = 16*) or in a condition. `printSquare`, however, only prints to the screen. Its result cannot be used in another computation. `printSquare(3) + printSquare(4)` does not even compile, because `printSquare` is `void`.

Pure methods are easier to test. It is enough to check `square(4) == 16`; there is no need to capture screen output or to remember the earlier value of a field. This is why a computing method and a printing method are kept apart. The computation is done in a pure method, and the printing is left to

a separate method or to `main`. When a side effect is unavoidable, for example when a report really must be written, the side effect is named explicitly and confined: the name `writeReport` says what it does, and it does nothing else.

Printing to the screen is not the only thing that spoils the purity of a method. A method that changes a `static` field also has a side effect. The result of a method that increases a counter on every call may vary with how many times it has been called. Methods that read from the keyboard or generate random numbers are also not pure, because they can give different results when called with the same input.

9.13 Summary

- A **method** is a named group of statements defined inside a class that does a task. It takes its input through **parameters** and gives its result as a **return value**. A method is defined once and **called** wherever you like and as many times as you like.
- The **formal parameters** written in the definition are variables; the **actual arguments** written in the call are values. Arguments are matched to parameters in order; their number and types must fit.
- In a call, the arguments are evaluated and copied into the parameters, the body runs, `return` sends the value, and the calling code continues with the value that has taken the place of the call.
- Each call opens its own **frame** on top of the **call stack**. Only the topmost frame runs; when the method ends, its frame is deleted together with all its local variables. The last frame opened is the first to close.
- The header consists of the access modifier, the modifier, the return type, the name and the parameter list. The **signature** is the name and the list of parameter types; the return type is not part of the signature.
- `main` is `static`, because the JVM calls it when no object exists. A `static` method cannot directly call a non-`static` method.
- A `void` method does not return a value; its job is a **side effect**. A `void` call cannot be part of an expression. **Early return** filters out special cases at the start of the method.
- In a method that is not `void`, every path must end with a `return`. The returned value must fit the return type; narrowing is not done automatically.
- Java always **passes arguments by value**. Even if a method changes its parameter, the caller's variable does not change; this is why a `swap` method that swaps primitive variables cannot be written. The new value is returned and the caller assigns it.
- In **overloading**, methods with the same name have different parameter lists. The compiler makes the choice at compile time, looking first for an exact match and then for the closest widening. Two methods that differ only in return type, and two equally fitting candidates, are compile errors.
- **Variable-length arguments** (`int... s`) are an array inside the method. The parameter comes at the very end of the parameter list; an empty array is passed in a call with no arguments.

- **Scope** is the region in which a variable can be used, and it is bounded by curly braces. Two local variables with the same name cannot be declared in the same scope. A local variable with the same name as a field **shadows** the field. A local variable cannot be read before it has been given an initial value.
- `Math` methods are `static` and overloaded. `sqrt` and `pow` return `double`, and `round(double)` returns `long`.
- A good method does a single task, has a meaningful name, takes few parameters, returns its result and can be tried with independent examples. **Abstraction** separates what is done from how it is done; **decomposition** splits a big task into submethods; a **contract** states the preconditions and postconditions.
- A method can call another method (`lcm` → `gcd`, `max3` → `max`); each call opens its own frame on the stack. A **pure method** always gives the same result for the same input and has no side effects; this is why it is easy to test and can be reused everywhere.

9.14 Key terms

These are the key terms introduced in this chapter.

method method call parameter formal parameter argument actual argument return type return value return method signature static method main method command-line argument stack stack frame call stack code tracing early return side effect pass by value reference method overloading widening conversion narrowing conversion varargs enhanced for loop scope local variable field shadowing compile-time error abstraction decomposition contract reuse pure method integer division overflow identity element greatest common divisor (GCD) least common multiple (LCM)

9.15 Test Yourself

1. The method `static double circleArea(double r) { return Math.PI * r * r; }` is called in `main` as `double s = circleArea(2 * radius);`. What are the parameter, the argument, the return type and the signature in these lines?

   Answer

   The formal parameter is `r`; it is the variable declared with its type (`double r`) in the header. The actual argument is the expression `2 * radius`; its value is computed first and this value is copied into r. The return type is `double`. The signature is `circleArea(double)`; the return type and the parameter name are not part of the signature.

2. Three methods are defined as follows: `a()` first prints "a1 ", then calls `b()`, then prints "a2 ". `b()` prints "b1 ", calls `c()`, prints "b2 ". `c()` only prints "c ". `main` calls `a()`. What is the output? What is the largest number of frames on the stack?

   Answer

   The output is `a1 b1 c b2 a2`. Each method waits until the method it called finishes and then continues where it left off. While `c` is running there are 4 frames on the stack: `main`, `a`, `b` and `c`. The frames close in the reverse of the order in which they were opened: first `c`, then `b`, and `a` last.

3. For the method `static void zeroOut(int n) { n = 0; }`, what does `int k = 9; zeroOut(k); System.out.println(k);` print? How should the method and the call be changed to really zero k?

   Answer

   It prints 9. Java passes by value: n is a copy of k, and `n = 0` changes only the copy. When the method ends, n is deleted. The method is written to return a value, `static int zeroOut() { return 0; }`, and the caller does the assignment itself: `k = zeroOut();`. In this example, however, the simplest solution is not to use a method at all and write `k = 0;`.

4. A class has only `static void g(long x)` and `static void g(double x)`; both print the type and value of their parameter. Which method do the calls `g(5)`, `g(5.5f)` and `g('x')` run?

   Answer

   There is no exact match for any of the calls; the compiler chooses the closest widening. `g(5)`: `int` → `long` is closer than to `double`; `long 5` is printed. `g(5.5f)`: `float` cannot be narrowed to `long` but widens to `double`; `double 5.5` is printed. `g('x')`: the `long` version is chosen through the widening `char` → `int` → `long`; since the code of the character 'x' is 120, `long 120` is printed.

5. For `static int count(int... s) { return s.length; }`, what do `count()`, `count(5)`, `count(new int[]{1, 2, 3})` and `count(1, 2)` return? Why does the header `static int count(int... s, int k)` not compile?

   Answer

   0, 1, 3 and 2, respectively. In the call with no arguments an empty array is passed (its length is 0); if an array is given directly, that array is used. In the second header the varargs parameter is not at the end of the list; the compiler gives the error `varargs parameter must be the last parameter`, because it cannot know which of the values in the call goes to k.

6. Which line of the following method gives a compile error, and why? `static int`

`sumOfSquares(int n) { for (int i = 1; i <= n; i++) { int sq = i * i; } return sq; }`

Answer

The line `return sq;` gives a `cannot find symbol` error. `sq` is declared in the body block of the `for` loop, and its scope ends when the block closes; outside the loop there is no variable with this name. The method also does not achieve its purpose: the squares are not being accumulated. The correct form is to declare `int total = 0;` before the loop, write `total += i * i;` in the body, and finish with `return total;`.

7. In how many iterations does `gcd(105, 30)` finish with Euclid's algorithm, and what does it return? What does the `lcm` method return for the same numbers?

Answer

Iteration 1: *105 % 30 = 15* (105 = 3 × 30 + 15); a = 30, b = 15. Iteration 2: *30 % 15 = 0*; a = 15, b = 0. The condition becomes false and 15 is returned after 2 iterations. *lcm = 105 / 15 × 30 = 7 × 30 = 210*. Check: *210 = 2 × 105 = 7 × 30*.

8. In the call `isPrime(91)`, how many iterations does the loop run and what is returned? How many iterations does `isPrime(37)` run?

Answer

91 = 7 × 13. For i = 2, 3, 4, 5, 6 the condition (*i ≤ 91 / i*) is true and the remainder is not 0. At i = 7, *7 ≤ 91 / 7 = 13* is true and *91 % 7 = 0*: `false` is returned in the 6th iteration. For 37, i = 2…6 is tried (5 iterations); at i = 7, *7 ≤ 37 / 7 = 5* is false and `true` is returned.

9. What do `Math.round(3.5)`, `Math.round(-3.5)`, `Math.abs(-0.5)` and `Math.max(-1, -8)` return? Why does `int r = Math.round(3.5);` not compile?

Answer

4, −3, 0.5 and −1. Numbers exactly at the half are rounded to the larger value: 3.5 → 4, −3.5 → −3. `Math.abs` returns `double` for a `double` argument. −1 is greater than −8. Because `Math.round(double)` returns `long`, the result cannot be assigned directly to an `int` (`possible lossy conversion from long to int`); `int r = (int) Math.round(3.5);` must be written.

10. For `static int triple(int x) { return 3 * x; }`, what do `triple(triple(2))` and `triple(2) * triple(1)` give? If the same method printed with `System.out.println(3 * x)`

instead of returning the result, could these two expressions be written?

Answer

The first: the inner call gives 6, the outer *triple(6) = 18*. The second: *6 × 3 = 18*. Both are 18. If the method were a `void` method that printed, neither expression would compile: a `void` call can be neither an argument to another call nor an operand of a multiplication. A pure method that returns its result can be used in any expression.

9.16 Further reading

- Oracle, *The Java Tutorials*: the pages “Defining Methods” and “Passing Information to a Method or a Constructor”; method declaration, overloading, varargs and pass by value are explained with short examples.
- R. Sedgewick and K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), Section 2.1 “Static Methods”: `static` methods, scope and modular programming with methods, worked through with examples from scientific computing.
- A. B. Downey and C. Mayfield, *Think Java* (2nd ed., O'Reilly, 2019): methods, parameters and return values are explained with stack diagrams for people new to programming; the online version of the book is free.
- Oracle, *Java Language Specification, Java SE 21*, §8.4 “Method Declarations” and §15.12 “Method Invocation Expressions”: the exact rules for the signature, return, and how the compiler chooses among overloads.

9.17 References

1. P. Deitel, H. Deitel, *Java: How to Program, Early Objects*, 11th ed., Pearson, 2018.
2. Y. D. Liang, *Introduction to Java Programming and Data Structures*, 13th ed., Pearson, 2024.
3. Oracle, *The Java Tutorials: Defining Methods; Passing Information to a Method or a Constructor*.
4. Oracle, *Java Language Specification, Java SE 21 Edition*, Chapter 8: Classes (§8.4 Method Declarations).
5. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 2.1 Static Methods.
6. A. B. Downey, C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019.

Arrays

Declaring 40 separate variables to store the grades of 40 students in a class is both tedious and error-prone, and when the number of grades changes the program has to be rewritten. An array solves this problem: it holds many values of the same type under a single name, in numbered cells. These notes begin with the structure of an array in memory. They then cover declaring arrays, reaching elements by index, and traversing an array with loops. Basic operations such as the sum, the average, the largest and smallest values, reversing, counting frequencies and the sieve of Eratosthenes are traced step by step. Copying, the `Arrays` class, passing arrays to methods, and how searching and sorting are written in Java follow. The last sections deal with two-dimensional arrays, matrix operations and common mistakes with arrays. All the programs shown in the notes were compiled and run with JDK 21, and the console output and error messages are taken verbatim from those runs.

LEARNING OUTCOMES

Creation
Declare arrays and create them with `new` and with an initializer; explain that an array variable holds a reference and that cells receive default values.

Access
Reach elements by index and use the `length` field correctly; recognize the cause and the message of an out-of-bounds error.

Processing
Traverse an array with the classic and the enhanced `for`; write sum, average, maximum–minimum, reversing, counting, copying, searching and sorting in Java and state their costs.

Matrices
Do row–column operations with two-dimensional arrays; explain jagged arrays and the shallow-copy trap; code the transpose and matrix multiplication.

10.1 The concept of an array

In the programs so far, each value has lived in its own variable: `int grade1`, `int grade2`, `int grade3`. For three grades this works. For forty grades, forty variables are needed, and finding their average means writing a sum with forty terms. If the number of grades rises to forty-one, the program has to be changed. The source of the problem is that the variables are independent of one another. The program cannot say “the third grade” or “the next grade”, because variable names are text, not numbers.

10.1.1 What is an array?

Definition 10.1

An **array** is a data structure that stores a fixed number of values of the same type in a single object, in consecutive **cells**. The values in the cells are called **elements**. Each element is reached by its **index**, the sequence number of its cell. In Java, indices start at 0. The number of cells is the **length** of the array; the length is fixed when the array is created and cannot change afterward.

Each of the three conditions in the definition has a reason. The **same type** condition makes every cell the same size: each cell of an `int` array holds 4 bytes. The **fixed number** condition allows the array to occupy memory as one contiguous block. **Consecutive cells** make it possible to find the location of any element with a single calculation.

In Java, an array is an **object**. Objects are created in the region of memory called the **heap**. The local variables of methods, in contrast, are kept in a separate region called the **stack**. An array variable lives on the stack, but it does not contain the elements. The variable stores the **reference** to the array object on the heap, that is, a value that shows where the object is in memory. In the figures a reference is drawn as an arrow.

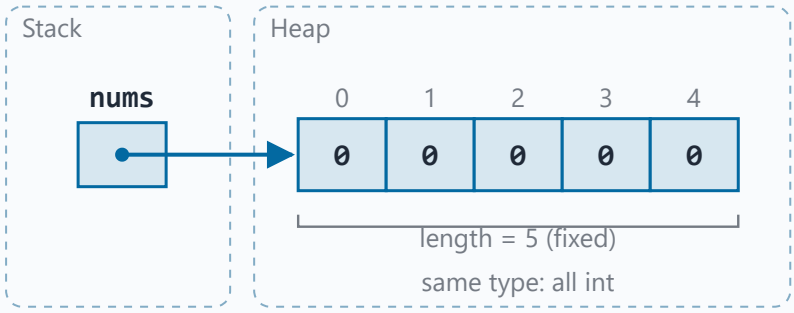


Figure 10.1: Memory after `int[] nums = new int[5];`. The variable `nums` on the stack points to the five-cell array object on the heap. The cells are numbered 0 to 4, all have the same type (`int`), and all start with the

value 0. The length is 5 and does not change.

Figure 10.1 shows this separation. The box on the left is the variable itself; it contains only an arrow, that is, a reference. The five cells on the right are the array object. At first this separation looks like an unnecessary detail. Yet most of the surprising behavior of arrays comes from it. Two variables can point to the same array; when an array is passed to a method, the method can change the original array; and the `==` operator compares the references of two arrays, not their contents. Each of these behaviors is worked through with an example in later sections.

10.1.2 Index and address calculation

The elements of an array are reached by writing the index in square brackets. `a[0]` is the first element and `a[1]` the second. In an array of five elements, the last element is `a[4]`. The general rule is this: for an array of length `n`, the first index is 0 and the last is `n - 1`. In Java the length of an array is read with `a.length`, so the last element can always be written as `a[a.length - 1]`.

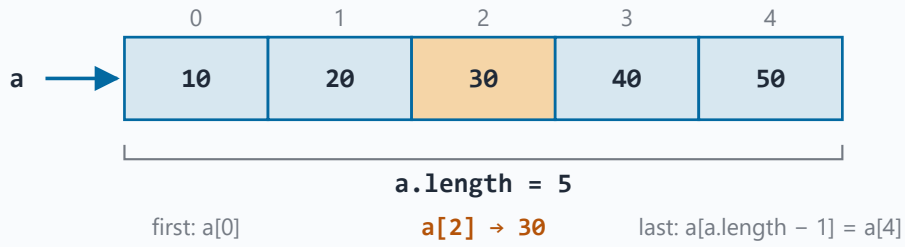


Figure 10.2: The array `a` with five elements: 10, 20, 30, 40, 50. Indices run from 0 to 4 and `a.length` is 5. The first element is `a[0]` and the last is `a[a.length - 1]`, that is, `a[4]`. The highlighted cell is `a[2]`, and its value is 30.

Starting the index at 0 feels strange to beginners. In everyday life, counting starts at 1. In Java, in C and in many other languages, however, the index is really an **offset**: it says how many cells to move forward from the start of the array. The first element is at the start, so its offset is zero. `a[2]` is two cells away from the start.

This interpretation also explains why access is fast. Because an array consists of consecutive cells in memory, the address of cell number `i` is computed with a single formula:

> address(a[i]) = start address + i × cell size

Example 10.1

Suppose the start address of an `int` array on the heap is 1000. Since an `int` takes 4 bytes, the cell size is 4. What is the address of `a[3]`?

In the formula `i` = 3 and the cell size is 4: 1000 + 3 × 4 = 1000 + 12 = 1012. The processor goes directly to this address and reads the value. The cells `a[0]`, `a[1]` and `a[2]` are never looked at. The same calculation for `a[999]` is also a single multiplication and a single addition: 1000 + 999 × 4 = 4996.

The size of the calculation does not depend on the length of the array. An array of five elements and one of a million elements both need one multiplication and one addition. This is why the cost of access by index is **constant time**, written **O(1)** in big-O notation. Big-O notation summarizes how the running time of an operation grows as the input grows. O(1) means "the same time however large the input", while O(n) means "the time grows in proportion to the number of elements". In a real Java program the start address is not shown to the programmer, and the JVM additionally checks on every access that the index is valid. But the principle of the calculation is the same.

10.1.3 Strengths and weaknesses of arrays

The structure of an array gives it some advantages and also sets some limits. Both come from the same source: fixed-size, consecutive cells.

Advantage	Limit
Fast access by index	The size is fixed
Element type is consistent	Inserting in the middle is costly
Easy to process with loops	Going out of bounds is a runtime error
Orderly layout in memory	Holds only one element type

Table 10.1: Advantages and limits of arrays.

In the table, "element type is consistent" and "holds only one element type" are two sides of the same property. The compiler prevents text from being put into an `int` array by mistake, which is a guarantee. On the other hand, it is impossible to keep a student's name, number and grade in the same array; that takes three separate arrays or another structure.

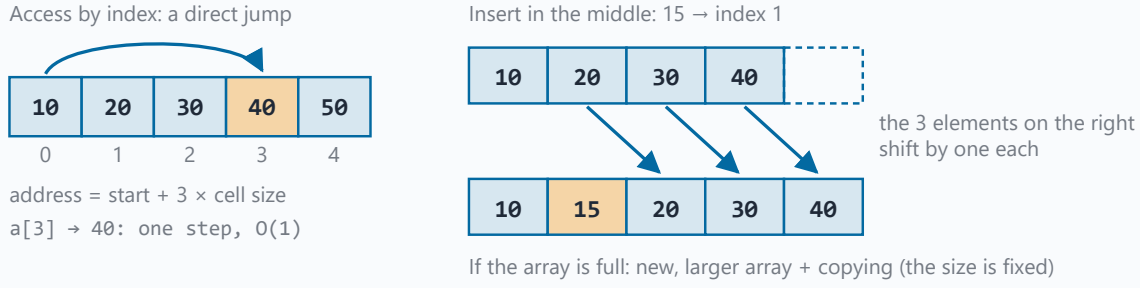


Figure 10.3: Left: access by index, where `a[3]` is reached in one step by an address calculation. Right: inserting in the middle, where 20, 30 and 40 are each shifted one cell to the right to put 15 at position 1. If the array is full, a new, larger array must be created and all the elements copied.

The right half of Figure 10.3 shows why inserting in the middle is expensive. Suppose 15 is to be placed between 10 and 20 in the array `{10, 20, 30, 40}`. Because the cells are consecutive, there is no operation “slip a cell in between”. 40 moves one place to the right, then 30, then 20, and 15 is written into the cell that has been freed, number 1. The number of elements shifted equals the number of elements after the insertion point. In the worst case, inserting at the front, all the elements move. This is why inserting and deleting in the middle cost O(n). Moreover, if there is no free cell at the end of the array, the insertion cannot be done at all: the length is fixed. A new, larger array is created and the old elements are copied into it.

10.2 Declaring and creating arrays

Using an array takes two separate jobs. First the array variable is declared, which opens a reference box on the stack. Then the array object is created and its reference is written into that box. The two jobs can be done on one line or on separate lines.

10.2.1 Declaration and new

In Java, an array type is written by adding a pair of square brackets after the element type: `int[]` means “array of int”, `String[]` “array of String” and `double[]` “array of double”. The array object is created with the `new` operator, and the length is given in square brackets. The following program shows four separate declarations together:

```
public class Declaration {
    public static void main(String[] args) {
        int[] nums = new int[5];
        String[] names = {"Ada", "Can", "Ece"};
        double[] readings;
        readings = new double[10];
        System.out.println(nums.length + " " + names.length + " " + readings.length);
    }
}
```

```
5 3 10
```

Let us examine the lines in order.

1. `int[] nums = new int[5];` does the declaration and the creation on one line. `new int[5]` creates an `int` array of five cells on the heap. The assignment writes the reference of this array into the variable `nums`.
2. `String[] names = {"Ada", "Can", "Ece"};` creates the array together with its values. No length is written; the compiler counts the values and makes the length 3.

3. `double[] readings;` only declares the variable. At this point there is no array on the heap.
4. `readings = new double[10];` creates the ten-cell array later and writes its reference into the variable.

The length does not have to be a constant; it can also be an `int` computed while the program runs. `new int[n]` creates as many cells as the current value of `n`. The length can be 0; such an array is called an **empty array** and has no cells. A negative length causes a `NegativeArraySizeException` when the program runs. Once an array has been created, its length cannot be changed. If `new double[20]` is later assigned to `readings`, the old array does not grow; the variable starts pointing to a new, separate array.

Common mistake

Declaring the variable and using it without creating the array. If `nums[0] = 7;` is written after the line `int[] nums;`, there are no cells. Java catches this at compile time:

```
Unassigned.java:4: error: variable nums might not have been initialized
        nums[0] = 7;
        ^
1 error
```

The message says "the variable nums might not have been assigned a value". The fix is an assignment such as `nums = new int[4];` before use.

10.2.2 Default values

The cells of an array created with `new` are not left empty. Java writes the **default value** of the element type into every cell. For numeric types this value is zero, and for `boolean` it is `false`. For object types, such as `String`, the default value is `null`. `null` is the special value meaning "this reference does not point to any object".

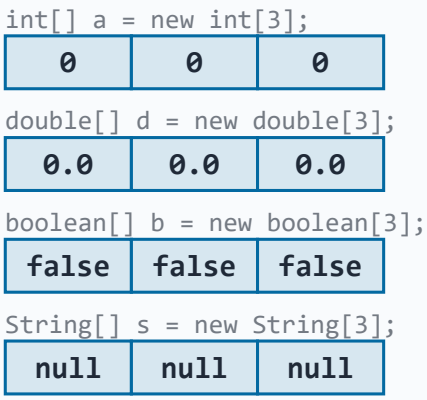


Figure 10.4: Initial values of three-cell arrays created with `new`: 0 for `int`, 0.0 for `double`, `false` for `boolean` and `null` for `String`.

The program below prints the values in Figure 10.4 to the console. The default value of a `char` array has been added:

```
public class Defaults {
    public static void main(String[] args) {
        int[] a = new int[3];
        double[] d = new double[2];
        boolean[] b = new boolean[2];
        String[] s = new String[2];
        char[] c = new char[2];
        System.out.println(a[1] + " " + d[0] + " " + b[1] + " " + s[0]);
        System.out.println((int) c[0]);
        System.out.println(a.length + d.length + s.length);
    }
}
```

```
0 0.0 false null
0
7
```

The first line prints one cell from each of four different arrays: 0, 0.0, `false` and `null`. The default value of a `char` array is the character whose code is 0. Because this character is invisible on the screen, it is printed as a number with an `(int)` cast. The last line contains a trap. The expression `a.length + d.length + s.length` contains no string, so `+` performs addition and the result is 3 + 2 + 2 = 7.

The default-value rule applies only to array cells and class fields. Java does not give a value on its own to local variables declared inside a method. `int x; System.out.println(x);` does not compile. By contrast, a cell of an array created with `new int[1]` always starts at 0.

10.2.3 The initializer

If the values of the elements are known in advance, the array is created by writing the values inside curly braces. This notation is called an **array initializer**: `int[] scores = {70, 85, 90};`. No length is written, because the compiler counts the values. Here the length is 3.

The short form of the initializer can be used only on the declaration line. It cannot be used to assign later to a variable that was declared earlier:

```
public class InitError {
    public static void main(String[] args) {
        int[] a;
        a = {1, 2, 3};
    }
}
```

```
InitError.java:4: error: illegal start of expression
        a = {1, 2, 3};
            ^
InitError.java:4: error: not a statement
        a = {1, 2, 3};
             ^
InitError.java:4: error: ';' expected
        a = {1, 2, 3};
              ^
3 errors
```

The compiler gives three messages for a single mistake. In such a flood of messages, the first message is the one to go by: “illegal start of expression”. The correct way is to put the array type in front of the initializer: `a = new int[]{1, 2, 3};`. This long form is also used to pass an array directly to a method: `search(new int[]{2, 4, 6, 8}, 5)`.

Java accepts both spellings of an array variable: `int[] a` and `int a[]`. The second is a leftover from the C language. These notes and the slides use the first form, because the type information (`int[]`) stays in one piece.

10.2.4 Creating an array step by step

The program below first declares an array, then creates it, writes values into two of its cells, and prints the lengths of two arrays:

```
public class Create {
    public static void main(String[] args) {
        int[] nums;
        nums = new int[4];
        nums[0] = 7;
        nums[3] = 2;
        String[] names = {"Ada", "Can", "Ece"};
        System.out.println(nums.length + " "
                           + names.length);
    }
}
```

```
4 3
```

Table 10.2 shows what is on the stack and the heap at each step of the program.

Line	Stack	Heap	Explanation
`int[] nums;`	`nums` = ?	—	Only the reference variable is created
`nums = new int[4];`	`nums` → array 1	array 1: [0, 0, 0, 0]	Four cells, all 0
`nums[0] = 7;`	same	array 1: [7, 0, 0, 0]	Follow the arrow, write 7 into cell 0
`nums[3] = 2;`	same	array 1: [7, 0, 0, 2]	Last cell: index `length - 1` = 3
`String[] names = {…};`	`names` → array 2	array 2: [Ada, Can, Ece]	Three values, length 3
`println(…)`	same	same	Output: `4 3`

Table 10.2: The trace table of the `Create` program. "?" marks a variable that has not been given a value yet.

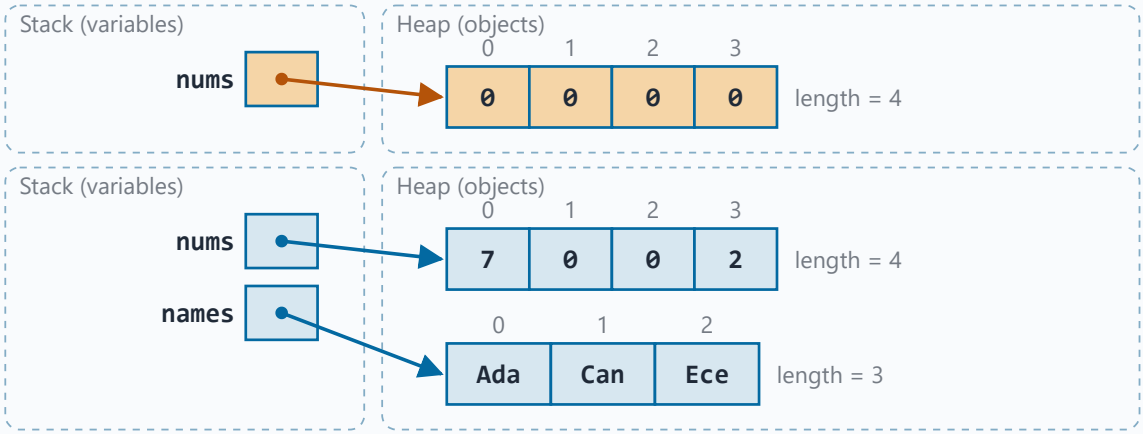


Figure 10.5: Two moments of the trace. Top: after `new int[4]`, four zero cells have been created on the heap and the reference has been written into the variable `nums`. Bottom: the end of the program, where `nums` points to the array [7, 0, 0, 2] and `names` to the array of three strings.

The first row of the table deserves attention: when `int[] nums;` runs, nothing happens on the heap. Figure 10.5 shows this emptiness and the change that follows. The `new` in the second line creates the object on the heap, and the assignment operator writes the reference into the variable. After that, every access such as `nums[0] = 7` follows the same path: the arrow in the variable is followed, the array on the heap is reached, the cell is found by its index and the value is written.

In the last line, `nums.length` and `names.length` are read. `length` is not a method but a **field** of the array, so it is written without parentheses. In the `String` class, by contrast, the length is obtained with the `length()` method. This difference is often confused: writing `a.length()` for an array or `s.length` for a string is a compile-time error. The `length` field is read-only; `nums.length = 10;` does not compile either.

10.3 Accessing elements

10.3.1 Reading and writing

An array element is used like an ordinary variable. If `scores[1]` appears inside an expression, the value in the cell is read. If it appears on the left of an assignment, the cell itself is meant and a new value is written into it. The index inside the square brackets does not have to be a constant; any expression that gives an `int` can be an index: `a[i]`, `a[i + 1]`, `a[a.length - 1]`.

```
public class Reading {
    public static void main(String[] args) {
        int[] scores = {70, 85, 90};
        int first = scores[0];
        scores[1] = 88;
        int last = scores[scores.length - 1];
        System.out.println(first + " " + last);
    }
}
```

```
70 90
```

Line	scores	first	last	Explanation
1	[70, 85, 90]	—	—	Three elements; indices 0, 1, 2
2	[70, 85, 90]	70	—	Reading: `scores[0]` → 70
3	[70, 88, 90]	70	—	Writing: 85 is erased, 88 is written
4	[70, 88, 90]	70	90	`scores.length - 1` = 3 − 1 = 2
5	[70, 88, 90]	70	90	Output: `70 90`

Table 10.3: The trace table of the `Reading` program.

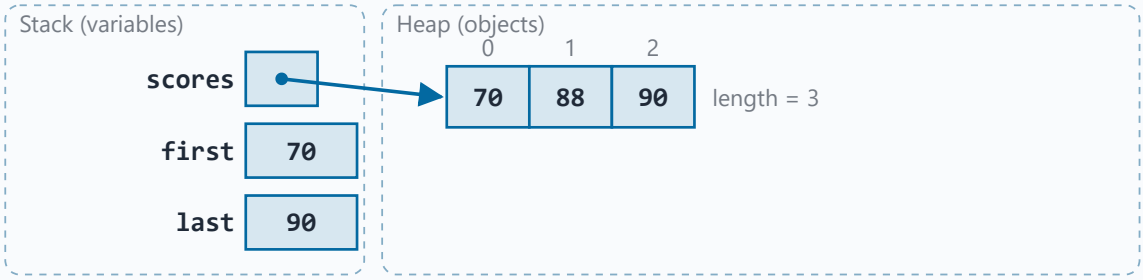


Figure 10.6: Memory at the end of the `Reading` program. `first` and `last` are ordinary `int` variables; their values were copied from the array. Cell 1 of the array now holds 88.

The expression `scores[scores.length - 1]` on line 4 is the general way to read the last element. First the inner expression is computed: 3 − 1 = 2. Then `scores[2]` is read. If the array had five elements, the same line would read `scores[4]` without any change. Writing the last index by hand (`scores[2]`), on the other hand, silently gives the wrong element when the array grows.

Figure 10.6 also shows an important detail: `int first = scores[0];` copies the value out of the array. After this line, even if `scores[0]` changes, `first` stays 70. For primitive types (`int`, `double`, `boolean`, `char`), assignment always copies the value.

10.3.2 The out-of-bounds error

Valid indices are the integers from 0 to `length - 1`. A negative index, or `length` itself, is invalid. Access with an invalid index stops the program in Java with an `ArrayIndexOutOfBoundsException`. This is an **exception**: an error condition that arises while the program is running and interrupts the normal flow.

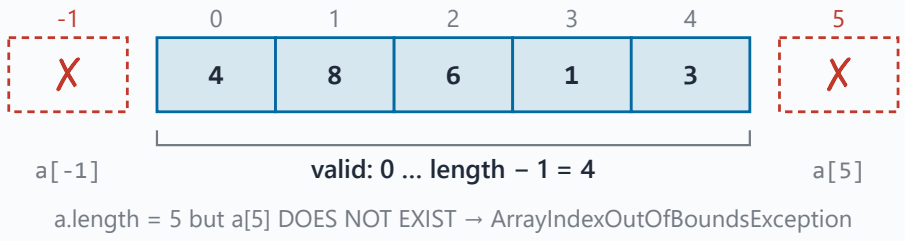


Figure 10.7: In an array of five elements the valid indices are 0…4. The cells `a[-1]` to the left of the array and `a[5]` to the right do not exist. `a.length` is 5, but there is no cell called `a[5]`.

Definition 10.2

In an array of length `n`, access with an index `i` that does not satisfy $0 \le i \le n - 1$ is an **out-of-bounds** access. Java checks this condition on every array access; if it is not satisfied, an `ArrayIndexOutOfBoundsException` is thrown.

The compiler usually cannot catch this error, because the index is often computed while the program runs. The following program compiles without any problem:

```
public class Negative {
    public static void main(String[] args) {
        int[] a = {4, 8, 6};
        System.out.println(a[-1]);
    }
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index -1 out of
bounds for length 3
        at Negative.main(Negative.java:4)
```

The message gives three pieces of information. The type of the error is `ArrayIndexOutOfBoundsException`. The detail part says “Index -1 out of bounds for length 3”: index −1 is outside the bounds of an array of length 3. The last line shows where the error occurred: line 4 of the file `Negative.java`. This check does not exist in languages like C, where an out-of-bounds write can silently corrupt a part of memory that belongs to another variable. Java's check has a cost, but it catches the error at exactly the right place.

10.3.3 One iteration too many: step by step

The most common source of out-of-bounds errors is writing `<=` instead of `<` in the loop condition. The error is called an **off-by-one** error:

```
public class Bounds {
    public static void main(String[] args) {
        int[] a = {4, 8, 6};
        for (int i = 0; i <= a.length; i++) {
            System.out.println(a[i]);
        }
    }
}
```

```
> java Bounds
4
8
6
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds
for length 3
        at Bounds.main(Bounds.java:5)
```

Iteration	`i`	`i <= a.length`	Action	Output
1	0	0 <= 3 → true	`a[0]` is printed	4
2	1	1 <= 3 → true	`a[1]` is printed	8
3	2	2 <= 3 → true	`a[2]` is printed	6
4	3	3 <= 3 → true	`a[3]` : no such cell	exception

Table 10.4: The trace table of the `Bounds` program (`a.length` = 3).

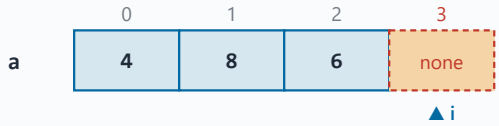


Figure 10.8: In the fourth iteration `i` = 3. Because the condition `3 <= 3` is true, the body is entered, but there is no cell number 3.

The first three iterations are fine and 4, 8, 6 are printed. In the fourth iteration `i` becomes 3. The condition `3 <= 3` is true, the body is entered and an attempt is made to read `a[3]` (Figure 10.8). The program stops at this point. Note that the error does not erase the earlier output: 4, 8 and 6 stay on the console. This is why a faulty program can look “almost right”.

The fix is to make the condition `i < a.length`. Then, when `i` is 3, `3 < 3` is false and the loop ends without entering the body. The rule to remember is short: if the index starts at 0, the condition is `< length`. `<= length - 1` is also correct, but longer and more error-prone.

10.4 Traversing with loops

The real power of an array appears when it is combined with a loop. Because the index can be a variable, a single loop can visit all the elements of the array in turn. Visiting every element of an array once is called **traversal**. Java has two ways to traverse: the classic `for` and the enhanced `for`.

10.4.1 The classic for

In the classic `for` loop, a counter, usually `i`, starts at 0 and increases by 1 on each iteration. Because the condition is `i < array.length`, the counter runs over exactly the valid indices:

```
public class Classic {
    public static void main(String[] args) {
        int[] scores = {70, 88, 90};
        for (int i = 0; i < scores.length; i++) {
          System.out.printf("%d: %d%n", i, scores[i]);
        }
    }
}
```

```
0: 70
1: 88
2: 90
```

`printf` does formatted printing: the first `%d` in the pattern is replaced by `i`, the second `%d` by `scores[i]`, and `%n` ends the line. The trace of the loop is given in Table 10.5.

`i`	`i < 3`	`scores[i]`	**Line printed**
0	true	70	`0: 70`
1	true	88	`1: 88`
2	true	90	`2: 90`
3	false	—	loop ends

Table 10.5: The trace table of the `Classic` program.

The last row is important: when `i` is 3, the condition is false and `scores[3]` is never accessed. The condition is tested at the start of every iteration, before the body. So it is no problem for `i` to reach an invalid value; the problem is entering the body with an invalid value.

The structure of the classic `for` is flexible. By changing the start, the condition and the step, the array can be traversed in different ways. `for (int i = a.length - 1; i >= 0; i--)` traverses the array from the end to the start. A step of `i += 2` advances by skipping one element. This flexibility comes from controlling the index directly.

10.4.2 The enhanced for

In many traversals the index itself is not needed; only the values of the elements are. Java has a shorter loop for these cases.

Definition 10.3

The **enhanced for** (also called **for-each**) is a loop written in the form `for (T x : array)`. The loop goes through the elements of the array in order from start to end; on each iteration it puts a **copy** of the next element into the variable `x`. The loop has no index.

The notation reads "for each `x` in the array". To the left of the colon is a variable with the element type, and to the right is the array. The word "copy" in the definition explains the behavior of the following program:

```
public class Enhanced {
    public static void main(String[] args) {
        int[] scores = {70, 88, 90};
        for (int score : scores) {
            score = score + 5;
            System.out.println(score);
        }
        System.out.println(scores[0]);
    }
}
```

```
75
93
95
70
```

Iteration	Cell copied	`score` (before → after)	`scores`	Output
1	`scores[0]`	70 → 75	[70, 88, 90]	75
2	`scores[1]`	88 → 93	[70, 88, 90]	93
3	`scores[2]`	90 → 95	[70, 88, 90]	95
last	—	out of scope	[70, 88, 90]	70

Table 10.6: The trace table of the `Enhanced` program. The array does not change at any step.

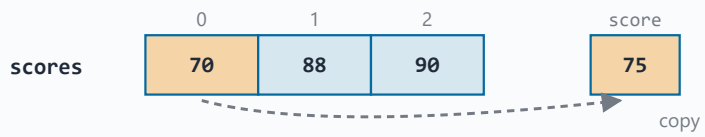


Figure 10.9: In the first iteration the 70 in `scores[0]` is copied into the variable `score`. `score = score + 5` only makes the copy 75; the cell in the array stays 70.

On each iteration `score` is increased by five and printed: 75, 93, 95. The last line prints 70. The array has not changed at all. Figure 10.9 shows why: `score` is not the cell itself but a separate variable. The value in the cell is copied into it at the start of each iteration. An assignment to the copy is not written back into the array. When the loop ends, the variable `score` also goes out of scope.

To really change the array, you must write into the cell by index. This calls for the classic `for`:

```
import java.util.Arrays;

public class AddFive {
    public static void main(String[] args) {
        int[] scores = {70, 88, 90};
        for (int i = 0; i < scores.length; i++) {
            scores[i] = scores[i] + 5;
        }
        System.out.println(Arrays.toString(scores));
    }
}
```

```
[75, 93, 95]
```

In this program, `scores[i]` is on the left of the assignment; the value is written directly into the cell. `Arrays.toString` prints all the elements of the array inside square brackets. This method is in the `java.util.Arrays` class and is imported at the top of the program.

Common mistake

Trying to update the array with the enhanced `for`. `for (int x : a) x = x * 2;` compiles and runs, but the array stays the same. The compiler does not even warn, because assigning to the loop variable is allowed by the language rules. The mistake is noticed only when the output is examined.

10.4.3 Which loop when?

The two loops cannot always be used in place of each other. The choice is decided by whether the job needs the index.

Job	Classic `for`	Enhanced `for`
Reading elements to add up, count or print	Works	Works and is shorter
Printing the position of an element	Works	Not possible (no index)
Writing a new value into a cell	Works	Not possible (the variable is a copy)
Looking at a neighboring element (`a[i + 1]`)	Works	Not possible

Job	Classic `for`	Enhanced `for`
Traversing backward or skipping	Works	Not possible (always start to end)
Traversing two arrays together with the same index	Works	Not possible

Table 10.7: The classic and the enhanced `for` compared.

The enhanced `for` is as safe as it is short: because there is no index, an out-of-bounds error cannot occur. For this reason it is preferred for traversals that only read. When an index is needed, switch to the classic `for`.

10.5 Basic array operations

The operations in this section are the building blocks of most programs written with arrays. In each of them the same question is asked: how many times must the array be traversed, and what information must be kept while traversing? All the examples run in O(n) time, because each element is looked at a constant number of times.

10.5.1 Sum and average

To find a sum, an **accumulator** variable is used. The accumulator starts at 0, the identity of addition, and each element is added to it:

```
public class Sum {
    public static void main(String[] args) {
        int[] nums = {4, 7, 2, 6};
        int sum = 0;
        for (int value : nums) sum += value;
        double avg = 0;
        if (nums.length > 0)
            avg = (double) sum / nums.length;
        System.out.println(sum + " " + avg);
        System.out.println(sum / nums.length);
    }
}
```

```
19 4.75
4
```

Step	`value`	`sum`	`avg`
Start	—	0	—
Iteration 1	4	0 + 4 = 4	—
Iteration 2	7	4 + 7 = 11	—

Step	`value`	`sum`	`avg`
Iteration 3	2	11 + 2 = 13	—
Iteration 4	6	13 + 6 = 19	—
`double avg = 0;`	—	19	0.0
`4 > 0` → division	—	19	19.0 / 4 = 4.75

Table 10.8: The trace table of the `Sum` program.

The average has two details. The first is the `(double)` cast. `sum` and `nums.length` are both `int`; the quotient of two `int`s is integer division, and the fractional part is discarded. The last line of the program shows this: `sum / nums.length` gives 4. The cast converts `sum` to `double` before the division; the division is then done in floating point and gives 4.75. The second is the `if (nums.length > 0)` check. In an empty array the length is 0. In floating-point division, 0.0 / 0 does not raise an error but produces `NaN` (Not a Number). In this case the check leaves the average at the safe starting value 0.0.

Cost: the loop does one addition per element; for n elements, n additions, that is O(n) time. Only two variables are used as extra memory; this means O(1) extra space.

10.5.2 Minimum and maximum

To find the smallest and largest elements, two **candidate** variables are kept. Each element is compared with the candidate; if the candidate is beaten, the element takes its place. The starting value of the candidates is the most critical point of this method:

```
public class MinMax {
    public static void main(String[] args) {
        int[] nums = {5, 3, 9, 1};
        int min = nums[0], max = nums[0];
        for (int value : nums) {
            if (value < min) min = value;
            if (value > max) max = value;
        }
        System.out.println(min + " " + max);
    }
}
```

```
1 9
```

Iteration	`value`	`value < min`	`value > max`	`min`	`max`
Start	—	—	—	5	5
1	5	5 < 5 → false	5 > 5 → false	5	5

Iteration	`value`	`value < min`	`value > max`	`min`	`max`
2	3	3 < 5 → true	3 > 5 → false	**3**	5
3	9	9 < 3 → false	9 > 5 → true	3	**9**
4	1	1 < 3 → true	1 > 9 → false	**1**	9

Table 10.9: The trace table of the `MinMax` program. Values that change are shown in bold.

The candidates are initialized with the first element of the array. A candidate is always a value that is really in the array, so the result is also an element of the array. Initializing a candidate with 0 looks tempting but is wrong. In an array whose elements are all positive, `min` is never updated and 0, which is not in the array, is found. In an array whose elements are all negative, the same problem appears with `max`.

Example 10.2

A method that finds the largest value starts its candidate at 0 and is called with the array `{-8, -3, -5}`. What is returned?

The candidate is 0. The tests −8 > 0, −3 > 0 and −5 > 0 are made in turn; all three are false. The candidate never changes and the method returns 0. The program run on JDK 21 also prints 0. The correct result is −3, and it is found when the candidate is started with `arr[0]`.

Another point worth noting in the program: the loop also examines the first element, although the candidates already started with that element. The comparisons in the first iteration (5 < 5, 5 > 5) come out false and do no harm. If you like, you can write the loop as a classic `for` starting at `i = 1`; the result does not change.

Common mistake

Accessing the first element of an empty array. `int min = nums[0];` finds no cell in an array of length 0:

```
0
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 0 out of
bounds for length 0
        at EmptyArray.main(EmptyArray.java:5)
```

The message says “index 0 is outside the bounds of an array of length 0”. In real programs the case `nums.length == 0` is handled separately first.

10.5.3 Reversing an array

The easy way to reverse an array is to create a new array and copy the elements into it from the end to the start. This way needs n cells of extra memory. A more economical way is to reverse the array **in place**: the elements at the two ends swap places, and then the ends move one step inward. Because two indices are used, the method is called the **two-pointer** technique.

```
import java.util.Arrays;

public class Reverse {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int left = 0, right = arr.length - 1;
        while (left < right) {
            int tmp = arr[left];
            arr[left] = arr[right];
            arr[right] = tmp;
            left++;
            right--;
        }
        System.out.println(Arrays.toString(arr));
    }
}
```

```
[5, 4, 3, 2, 1]
```

The swap is done with three assignments. If `arr[left] = arr[right]` were written directly, the value on the left would be lost; so it is first saved in the variable `tmp`. Then the value on the right is written to the left, and the saved value to the right.

Step	`left`	`right`	`left < right`	`tmp`	`arr`
Start	0	4	—	—	[1, 2, 3, 4, 5]
1st swap	0	4	true	1	[5, 2, 3, 4, 1]
The pointers move	1	3	—	—	[5, 2, 3, 4, 1]
2nd swap	1	3	true	2	[5, 4, 3, 2, 1]
The pointers move	2	2	false	—	[5, 4, 3, 2, 1]

Table 10.10: The trace table of the `Reverse` program.



Figure 10.10: The end of the first swap. The cells `left` = 0 and `right` = 4 have swapped: 5 has moved to the front and 1 to the back. `tmp` still holds 1.

In an array of five elements two swaps are made. In the third iteration both pointers point to cell 2; there is no need to move the middle element, and because `2 < 2` is false, the loop ends. In general, ⌊n/2⌋ swaps are made for an array of n elements; that is O(n) time. The extra memory is only three variables: O(1).

Common mistake

Doing the swap along the whole array. If `a[i]` and `a[a.length - 1 - i]` are swapped with `for (int i = 0; i < a.length; i++)`, the swaps done in the first half of the array are undone in the second half. The array `{1, 2, 3, 4}` ends up as `[1, 2, 3, 4]` again at the end of the program. The loop must run only up to the middle, with the condition `i < a.length / 2`.

10.5.4 Counting the elements above the average

Some questions cannot be answered in a single pass. “How many students scored above the average?” is one of them. To know whether a grade is above the average, the average is needed first. The average, in turn, can be found only after all the grades have been read. So the array is traversed twice. This is also a valuable property of an array: it stores the data and allows it to be read a second time.

```
public class AboveAverage {
    public static void main(String[] args) {
        int[] grades = {60, 75, 90, 45, 80};
        int sum = 0;
        for (int x : grades) sum += x;
        double avg = (double) sum / grades.length;
        int above = 0;
        for (int x : grades)
          if (x > avg) above++;
        System.out.println(avg + " " + above);
    }
}
```

```
70.0 3
```

x	**Pass 1:** sum	**Pass 2:** x > 70.0	above
60	60	false	0
75	135	true	1
90	225	true	2
45	270	false	2
80	350	true	3

Table 10.11: The trace table of the `AboveAverage` program. The first pass updates the sum, the second pass the counter.

At the end of the first pass the sum is 350, and the average is found to be 350 / 5 = 70.0. In the second pass each grade is compared with 70.0; 75, 90 and 80 are above it. The program prints `70.0 3`. The total cost of the two passes is 2n comparisons and additions. In big-O notation constant factors are not written; the cost is still O(n).

10.5.5 The second largest element

To find the second largest element of an array there is no need to sort it. Two candidates are kept in a single pass: `largest` and `second`. Repeated values make the question subtler: in the array `{7, 12, 5, 12, 9}` the largest is 12 and it occurs twice; the second largest element is 9.

```
public class SecondLargest {
    public static void main(String[] args) {
        int[] a = {7, 12, 5, 12, 9};
        int largest = Integer.MIN_VALUE;
        int second = Integer.MIN_VALUE;
        for (int x : a) {
          if (x > largest) {
            second = largest;
            largest = x;
          } else if (x > second && x != largest) {
            second = x;
          }
        }
        System.out.println(second);
        System.out.println(Integer.MIN_VALUE);
    }
}
```

```
9
-2147483648
```

Both candidates start at `Integer.MIN_VALUE`. This constant is the smallest value the `int` type can take, that is, −2147483648; the second line of the program's output shows it. This way the first element of the array, whatever it is, turns out larger than the candidates. For each element there are two cases:

1. If `x > largest`, a new largest has been found. The old largest is not wasted; it drops down to second place: first `second = largest`, then `largest = x`. If the order is reversed, the old value is lost.
2. Otherwise, if `x` is both greater than `second` and different from the largest, `x` is the new second. The condition `x != largest` prevents a repeat of the largest from being counted as the second.

`x`	**Case**	`largest`	`second`
—	Start	MIN	MIN
7	7 > MIN: new largest	7	MIN
12	12 > 7: new largest, 7 drops to second	12	7
5	5 is not greater than second (7)	12	7
12	12 == largest: a repeat, not counted	12	7
9	9 > 7 and 9 != 12: new second	12	9

Table 10.12: The trace table of the `SecondLargest` program. MIN stands for `Integer.MIN_VALUE`.

The array is traversed once: O(n) time, O(1) extra space. If the array is empty, has a single element, or all its elements are equal, for example `{4, 4}`, `second` is never updated and stays at `Integer.MIN_VALUE`. If after the loop `second` is still `Integer.MIN_VALUE`, there is no second distinct value in the array; in a real program this case should be reported separately as “no second largest”.

10.5.6 Counting frequencies: a dice histogram

To count how many times each of the six faces of a die comes up, six counters are needed. Instead of six separate variables and a long `if-else` chain, a single array of counters is used. The trick of the method is to **use the value as the index**: if the die shows 3, `count[3]` is increased by one.

```
public class Histogram {
    public static void main(String[] args) {
        int[] rolls = {3, 6, 3, 1, 6, 3};
        int[] count = new int[7];
        for (int die : rolls)
          count[die]++;
        for (int y = 1; y <= 6; y++) {
          System.out.print(y + ": ");
          for (int k = 0; k < count[y]; k++)
            System.out.print("*");
          System.out.println();
        }
    }
}
```

```
1: *
2:
3: ***
4:
5:
6: **
```

The counter array has 7 cells, because the die values are between 1 and 6 and cell number 6 is needed. Cell 0 is never used. One cell is wasted, but in return the notation `count[die]` reads directly. In the example the rolls are given as constants; in a real program they would be generated with `random.nextInt(6) + 1`.

The first loop counts the rolls. `count[die]++` is a single statement: first the value of `die` is read, the cell with that number is found and the value in the cell is increased by one. After the six rolls the counter array is `[0, 1, 0, 3, 0, 0, 2]`. The second loop prints one line for each face; the inner loop puts as many stars on that line as the count. Since 2, 4 and 5 never came up, those lines have no stars.

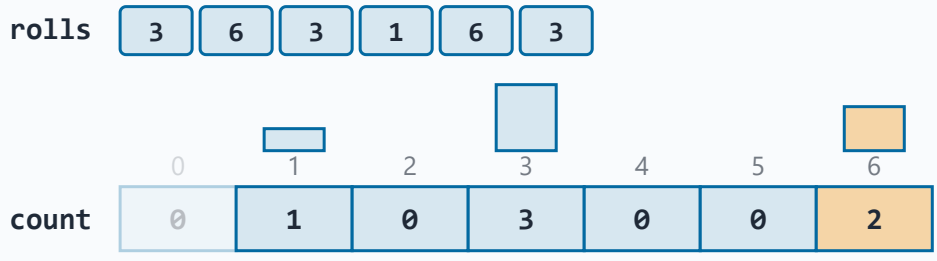


Figure 10.11: The counter array after all the rolls have been counted. The cell number is the die value; cell 0 is drawn faintly because it is not used. The bars above the cells are proportional to the counter values.

The counting loop does a single increment per roll; for m rolls, O(m) time. With an `if-else` chain, up to six comparisons would be needed for each roll. The same technique is used to find the frequency of letters or exam grades; the only condition is that the values correspond to small non-negative integers.

10.5.7 The sieve of Eratosthenes

An old and fast way to find all the prime numbers up to a given number is the sieve of Eratosthenes. The method does not search for primes; it eliminates the non-primes. It starts at 2. Every number that has not been eliminated is prime, and its multiples are eliminated. A `boolean` array is used to keep track of the eliminated numbers: if `composite[k]` is true, k has been eliminated.

```
public class Sieve {
    public static void main(String[] args) {
        int n = 20;
        boolean[] composite = new boolean[n + 1];
        for (int i = 2; i * i <= n; i++)
          if (!composite[i])
            for (int k = i * i; k <= n; k += i)
              composite[k] = true;
        for (int i = 2; i <= n; i++)
          if (!composite[i]) System.out.print(i + " ");
        System.out.println();
    }
}
```

```
2 3 5 7 11 13 17 19
```

The array has `n + 1` cells, because 20 is also used as an index. At the start all the cells are `false`; this is the default value of a `boolean` array and means “not yet eliminated”. The inner loop marks the multiples. The steps of the elimination were also traced on JDK 21, and the cells marked for each `i` are given in Table 10.13.

`i`	`i * i <= 20`	Case	Values of `k` marked
2	4 <= 20 → true	Not eliminated, prime	4, 6, 8, 10, 12, 14, 16, 18, 20
3	9 <= 20 → true	Not eliminated, prime	9, 12 (already), 15, 18 (already)
4	16 <= 20 → true	Eliminated; its multiples are not examined	—
5	25 <= 20 → false	The outer loop ends	—

Table 10.13: The steps of the elimination for n = 20. The word “already” shows that the cell had earlier been marked as a multiple of another prime.

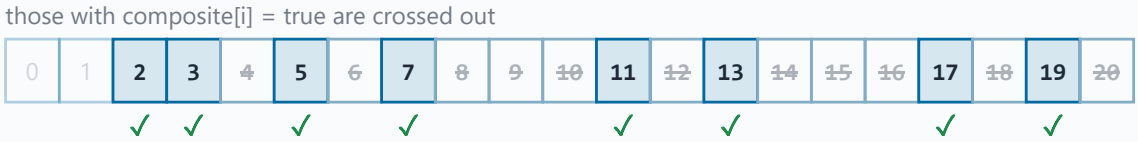


Figure 10.12: The `composite` array at the end of the elimination. The crossed-out numbers have been eliminated. The eight numbers with a check mark (2, 3, 5, 7, 11, 13, 17, 19) are prime.

The method has two speed-ups. The first is that multiples start at `i * i`. Smaller multiples of 3, such as 6, have already been eliminated as multiples of a smaller prime (2). The second is that the outer loop stops with the condition `i * i <= n`. Every non-prime number up to 20 has a divisor less than or equal to $\sqrt{20} \approx 4.47$. So there is no need to eliminate for 5 and beyond. When the same program is run with n = 100, it finds 25 primes.

The sieve shows that an array can serve both as data and as a “marking board”. Testing whether each number is prime by trial division is much slower than the sieve for large n. The time cost of the sieve is O(n log log n), which in practice grows very close to n. The price is a `boolean` array of n + 1 cells.

10.6 Copying, the Arrays class and methods

The fact that an array variable holds a reference matters most when copying. For a primitive type, `int y = x;` copies the value and the two variables become independent. For arrays, the same notation gives a completely different result.

10.6.1 Reference copy and element copy

The statement `int[] b = a;` does not create a new array. The reference in `a` is copied into `b`; now two variables point to the same object. This situation is called a **reference copy**. A change made through one is seen through the other, because there is only one array.

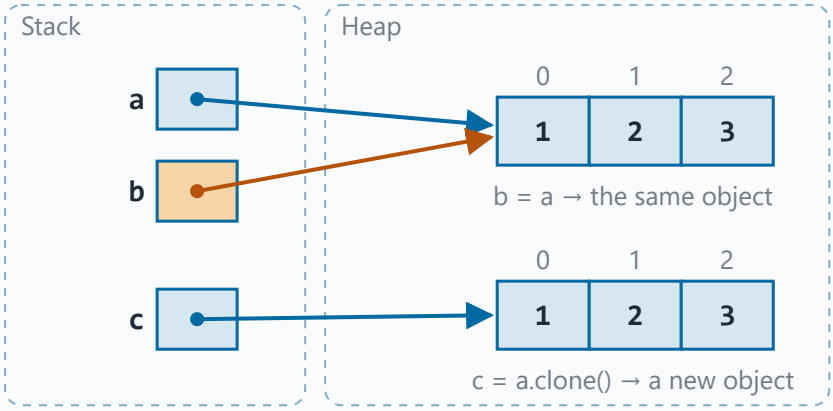


Figure 10.13: Reference copy and element copy. After `b = a` two arrows point to the same object. `c = a.clone()`, on the other hand, creates a new array on the heap and copies the elements into it.

For an independent copy, the elements of the array must be transferred to a new array. Java offers three ready-made ways to do this:

- `a.clone()`: creates a new array of the same length and copies all the elements.
- `Arrays.copyOf(a, a.length)`: does the same job; the second argument can also choose the length of the new array.
- `System.arraycopy(src, srcPos, dest, destPos, length)`: copies a given number of elements, starting from a given position, into an existing destination array.

The third way is the most flexible, but it requires the destination array to be created beforehand:

```
import java.util.Arrays;

public class ArrayCopy {
    public static void main(String[] args) {
        int[] a = {1, 2, 3, 4, 5};
        int[] d = new int[7];
        System.arraycopy(a, 1, d, 2, 3);
        System.out.println(Arrays.toString(d));
    }
}
```

```
[0, 0, 2, 3, 4, 0, 0]
```

The call `System.arraycopy(a, 1, d, 2, 3)` means "write 3 elements, starting at cell 1 of `a`, starting at cell 2 of `d`". The elements copied are 2, 3 and 4; they are written into cells 2, 3 and 4 of `d`. The remaining cells stay at the default value 0.

Definition 10.4

A **shallow copy** is a copy in which only the values in the cells of an array are copied. If the cells hold primitive values, a shallow copy is completely independent. If the cells hold references, only the references are copied; the objects they point to are shared between the two arrays.

`clone`, `Arrays.copyOf` and `System.arraycopy` make shallow copies. In `int` arrays this causes no problem, because the cells hold the numbers themselves. In arrays of objects and in two-dimensional arrays, however, the objects are shared, not copied. This trap is treated separately together with two-dimensional arrays.

10.6.2 Copying step by step

The program below shows the two kinds of copy side by side. The last line compares two references with `==`:

```
public class Copy {
    public static void main(String[] args) {
        int[] a = {1, 2, 3};
        int[] b = a;
        b[0] = 99;
        System.out.println(a[0]);
        int[] c = a.clone();
        c[1] = 50;
        System.out.println(a[1] + " " + c[1]);
        System.out.println((a == b) + " " + (a == c));
    }
}
```

```
99
2 50
true false
```

Line	`a`	`b`	`c`	Object 1	Object 2	Output
`int[] a = {1, 2, 3};`	→ 1	—	—	[1, 2, 3]	—	
`int[] b = a;`	→ 1	→ 1	—	[1, 2, 3]	—	
`b[0] = 99;`	→ 1	→ 1	—	[99, 2, 3]	—	
`println(a[0])`	→ 1	→ 1	—	[99, 2, 3]	—	99
`int[] c = a.clone();`	→ 1	→ 1	→ 2	[99, 2, 3]	[99, 2, 3]	
`c[1] = 50;`	→ 1	→ 1	→ 2	[99, 2, 3]	[99, 50, 3]	

Line	a	b	c	Object 1	Object 2	Output
`println(a[1] + " " + c[1])`	→ 1	→ 1	→ 2	[99, 2, 3]	[99, 50, 3]	2 50

Table 10.14: The trace table of the `Copy` program. "Object 1" and "object 2" are two separate arrays on the heap.

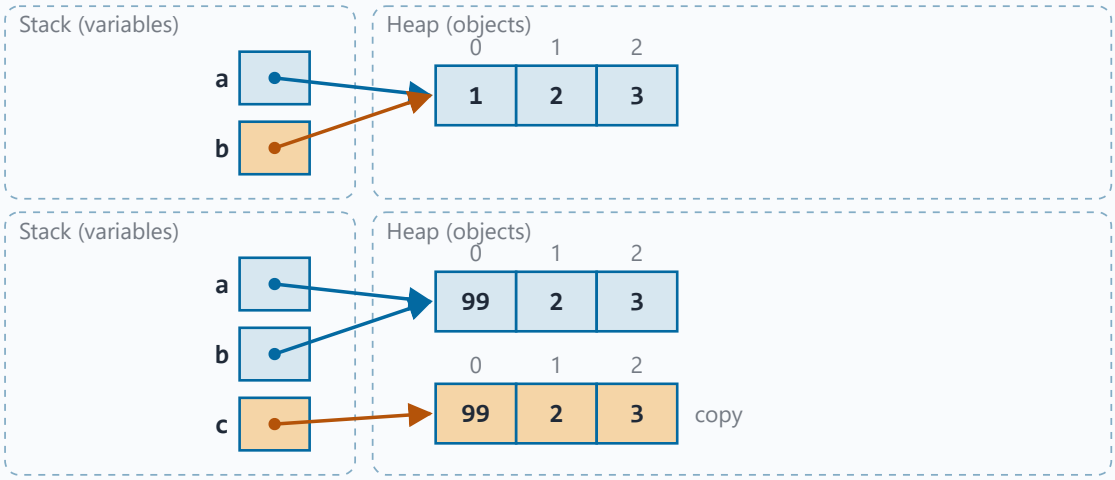


Figure 10.14: Top: after `b = a`, two arrows point to the same object. Bottom: after `a.clone()`, `c` points to a new object that is a copy of the current contents (99, 2, 3).

The line `b[0] = 99` seems to change only `b`. But the object `b` points to is also the object of `a`, and `a[0]` becomes 99. The `clone` call, on the other hand, copies the current contents into a new object. After that, a change made through `c` does not affect `a`. The output `true false` on the last line summarizes the two relationships. `a == b` is true, because the two variables hold the same reference. `a == c` is false; even though their contents are the same, they are different objects.

10.6.3 Arrays helpers

The `java.util.Arrays` class contains ready-made methods for frequent jobs with arrays. These methods are `static` and are called with the class name: `Arrays.toString(a)`. The line `import java.util.Arrays;` is added at the top of the program. The program below runs every row of the table in the slides on JDK 21:

```
import java.util.Arrays;

public class Helpers {
    public static void main(String[] args) {
        int[] a = {5, 2, 9}, r = {10, 20, 30, 40, 50}, p = {1, 2, 3}, q = {1, 2, 3}, f =
new int[4];
        int[][] m = {{1, 2}, {3, 4}};
        System.out.println(Arrays.toString(a));
        System.out.println(Arrays.deepToString(m));
        System.out.println(Arrays.toString(m).startsWith("[[I@"));
        Arrays.fill(f, 7);
        System.out.println(Arrays.toString(f));
        System.out.println((p == q) + " / " + p.equals(q));
        System.out.println(Arrays.equals(p, q));
        System.out.println(Arrays.toString(Arrays.copyOf(a, 5)));
        System.out.println(Arrays.toString(Arrays.copyOf(a, 2)));
        System.out.println(Arrays.toString(Arrays.copyOfRange(r, 1, 4)));
        System.out.println(a);
    }
}
```

```
[5, 2, 9]
[[1, 2], [3, 4]]
true
[7, 7, 7, 7]
false / false
true
[5, 2, 9, 0, 0]
[5, 2]
[20, 30, 40]
[I@548c4f57
```

Call	Result	Explanation
`Arrays.toString(a)`	`[5, 2, 9]`	Prints the elements of a one-dimensional array
`Arrays.deepToString(m)`	`[[1, 2], [3, 4]]`	Prints nested arrays too, expanding them
`Arrays.toString(m)`	`[[I@…`	For a 2D array, prints the references of the rows
`Arrays.fill(f, 7)`	`[7, 7, 7, 7]`	Writes the same value into all cells
`p == q` / `p.equals(q)`	`false / false`	Both compare references
`Arrays.equals(p, q)`	`true`	Compares the contents element by element
`Arrays.copyOf(a, 5)`	`[5, 2, 9, 0, 0]`	Longer copy: the extra cells are 0
`Arrays.copyOf(a, 2)`	`[5, 2]`	Shorter copy: truncated at the end
`Arrays.copyOfRange(r, 1, 4)`	`[20, 30, 40]`	Elements 1, 2 and 3; the end index is excluded

Table 10.15: Frequently used `Arrays` methods and their results in the `Helpers` program.

The last line of the program shows the result of printing an array directly with `println`: text that begins with `[I@`. The leading `[I` means “int array”. The hexadecimal number after the `@` sign is a code belonging to the object, and it can change on every run. This output gives no information about the elements. The same thing happens when `Arrays.toString` is applied to a two-dimensional array: the cells of the outer array are references to row arrays, and each is printed in the form `[I@…`. On this line the program only tests that the text begins with `[[I@` and prints `true`; the code itself is not shown because it changes from run to run.

`equals` needs separate attention. In `String`, `equals` compares contents. For arrays, `p.equals(q)` does the same job as `p == q` and looks only at the reference. To compare contents, use `Arrays.equals(p, q)`, or `Arrays.deepEquals` for two-dimensional arrays.

The range in `copyOfRange` follows a general Java habit: the start index is included and the end index is excluded. `copyOfRange(r, 1, 4)` takes elements 1, 2 and 3; the length of the copy is 4 − 1 = 3. The same rule holds in the `substring` method of the `String` class.

10.6.4 Arrays and methods

A method can take an array and return an array. In Java every argument passed to a method is passed by copying. For an array argument, what is copied is the reference, not the array itself. So the parameter in the method points to the **same** object as the array on the caller's side. Two rules follow:

1. If the method writes into cells through the parameter, the change is seen in the caller's array too.
2. If the method assigns a new array to the parameter, only the reference in its own local copy changes; the caller's variable is not affected.

```
import java.util.Arrays;

public class Methods {
    static void doubleAll(int[] arr) {
        for (int i = 0; i < arr.length; i++)
            arr[i] = arr[i] * 2;
    }

    static void assignNew(int[] arr) {
        arr = new int[]{0, 0, 0};
    }

    static int[] squares(int n) {
        int[] result = new int[n];
        for (int i = 0; i < n; i++)
            result[i] = (i + 1) * (i + 1);
        return result;
    }

    public static void main(String[] args) {
        int[] a = {1, 2, 3};
        doubleAll(a);
        System.out.println(Arrays.toString(a));
        assignNew(a);
        System.out.println(Arrays.toString(a));
        System.out.println(Arrays.toString(squares(4)));
    }
}
```

```
[2, 4, 6]
[2, 4, 6]
[1, 4, 9, 16]
```

In the call `doubleAll(a)`, the parameter `arr` points to the object of `a` and writes into its cells; `a` becomes `[2, 4, 6]`. `assignNew(a)`, on the other hand, binds a new object to `arr`. This assignment affects only the method's own variable; `a` keeps pointing to the old object and stays `[2, 4, 6]`. The method `squares(4)` creates a new array and returns its reference. When the method ends, the local variable `result` disappears, but the array on the heap lives on, because its reference has been given to the caller.

This behavior has a side effect. A method that sorts, reverses or fills an array changes the caller's data permanently. If the original order must be kept, the method should first take a copy. This rule comes up again in the sorting and median examples. Incidentally, the parameter of the `main` method is also an array: `String[] args` holds the arguments given on the command line, and `args.length` gives the number of arguments.

10.7 Searching and sorting

The idea, pseudocode and detailed traces of the algorithms linear search, binary search, bubble sort and insertion sort are described in the Algorithms lecture notes. This section focuses on writing the same algorithms with Java arrays, on short traces, and on points specific to arrays: index bounds, in-place changes and the ready-made methods of the `Arrays` class.

10.7.1 Linear search

Linear search traverses the array from start to end and compares each element with the target. At the first match it returns the index; if the target is absent it returns −1. Since −1 cannot be a valid index, it is used to mean “not found”.

```
public class Linear {
    static int search(int[] arr, int target) {
        for (int i = 0; i < arr.length; i++)
            if (arr[i] == target) return i;
        return -1;
    }

    public static void main(String[] args) {
        int[] d = {8, 3, 6, 2};
        System.out.println(search(d, 6));
        System.out.println(search(d, 5));
    }
}
```

```
2
-1
```

`i`	`arr[i]`	`== 6 ?`
0	8	false
1	3	false
2	6	true → `return 2`

Table 10.16: The trace of the call `search(d, 6)`.

Two points stand out in the Java version. First, `return i` is inside the loop; the method ends at the first match and the remaining elements are not looked at. `return -1` is outside the loop and runs only after all the elements have been tried. Putting this line inside the loop by mistake, in an `else`

branch, would end the search at the first element. Second, the `arr` parameter points to the same array as `d`; no element is copied during the call. Since 5 is not in the array, the second call returns −1 after four comparisons. In the worst case n comparisons are made: O(n).

10.7.2 Arrays.sort and Arrays.binarySearch

Java provides ready-made methods for sorting and binary search. `Arrays.sort(a)` sorts the array from smallest to largest. `Arrays.binarySearch(a, x)` searches a sorted array for `x` using binary search.

```
import java.util.Arrays;

public class SortDemo {
    public static void main(String[] args) {
        int[] a = {5, 2, 9, 1, 7};
        Arrays.sort(a);
        System.out.println(Arrays.toString(a));
        System.out.println(Arrays.binarySearch(a, 7));
        System.out.println(Arrays.binarySearch(a, 6));
    }
}
```

```
[1, 2, 5, 7, 9]
3
-4
```

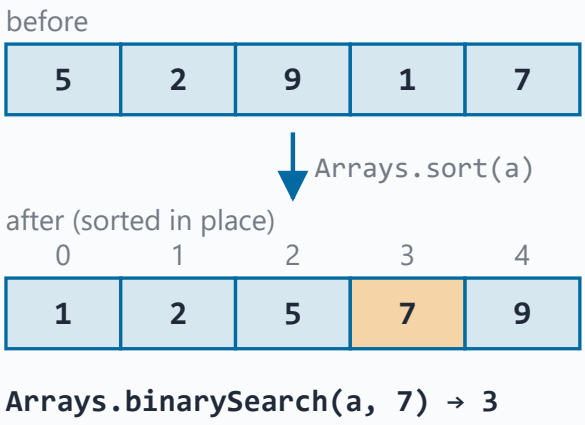


Figure 10.15: `Arrays.sort` sorts the array in place: the cells of the same object are rearranged. In the sorted array the index of 7 is 3.

`Arrays.sort` does not return a new array; it changes the array itself **in place** (Figure 10.15). The original order is lost. If the original order is needed for another job, a copy is taken first and the copy is sorted.

If `Arrays.binarySearch` finds the target, it returns its index. If it does not, it returns a negative value: `-(insertion point) - 1`. The **insertion point** is the position at which the target would go in the sorted array. In the sorted array `[1, 2, 5, 7, 9]`, 6 would go between 5 and 7, that is, at position 3. The returned value is −3 − 1 = −4. The "− 1" in the formula makes the result negative even when the insertion point is 0; otherwise −0 and 0 could not be told apart.

Common mistake

Calling `Arrays.binarySearch` on an unsorted array. The method assumes the array is sorted and does not check it. The result can be meaningless: even for a value that is in the array, the answer may be "not present". A question in the Test Yourself section shows this on a concrete array.

Both methods are in the `java.util.Arrays` class and require `import java.util.Arrays;`. For arrays of primitive types, `Arrays.sort` uses a kind of quicksort that runs in O(n log n) time on average. For large arrays this is much better than the $O(n^2)$ cost of bubble sort and insertion sort.

10.7.3 The median

Definition 10.5

The **median** is the value exactly in the middle of a sorted array. If the number of elements n is odd, the median is `s[n / 2]`. If n is even, it is the average of the two middle elements, `s[n/2 - 1]` and `s[n/2]`.

An advantage of the median over the average is that it is not affected by extreme values. The average of the array `{1, 2, 3, 4, 100}` is 110 / 5 = 22.0; this value is close to none of the elements. The median is 3. In data such as income, house prices or response times there are a few extreme values, so the median gives a more meaningful "typical value".

```java
import java.util.Arrays;

public class Median {
    public static void main(String[] args) {
        int[] a = {7, 3, 9, 1, 5};
        int[] s = Arrays.copyOf(a, a.length);
        Arrays.sort(s);    // a is untouched
        int n = s.length;
        double m = (n % 2 == 1) ? s[n / 2]
            : (s[n/2 - 1] + (double) s[n/2]) / 2;
        System.out.println(m);   // 5.0
        System.out.println(Arrays.toString(a) + " " + Arrays.toString(s));
    }
}
```

```
5.0
[7, 3, 9, 1, 5] [1, 3, 5, 7, 9]
```

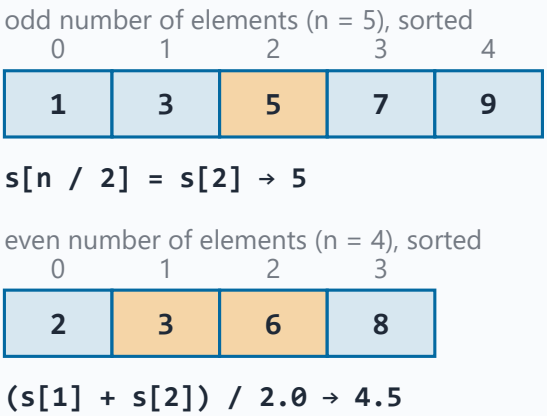


Figure 10.16: Top: an odd number of elements (n = 5): the median is `s[2]` = 5. Bottom: an even number of elements (n = 4): the median is (`s[1]` + `s[2]`) / 2.0 = (3 + 6) / 2.0 = 4.5.

The program has three points specific to arrays. First, the sorting is done on a copy: `s` is created with `Arrays.copyOf` and the original order of `a` is preserved. The second line of the program's output confirms this: `a` is still `[7, 3, 9, 1, 5]`. Second, the conditional operator (`? :`) combines the odd and even cases in a single expression. Third, in the even case `s[n/2]` is cast with `(double)` before the addition. This way both the addition and the division are done in `double`: the sum of two large `int`s cannot overflow the `int` limit, and the quotient comes out fractional. Without the cast, the quotient of two `int`s would be taken, and (3 + 6) / 2 would give 4 instead of 4.5. The program below verifies this difference and the extreme-value example on JDK 21:

```
import java.util.Arrays;

public class Median2 {
    static double median(int[] a) {
        int[] s = Arrays.copyOf(a, a.length);
        Arrays.sort(s);
        int n = s.length;
        return (n % 2 == 1) ? s[n / 2] : (s[n/2 - 1] + (double) s[n/2]) / 2;
    }

    public static void main(String[] args) {
        int[] u = {1, 2, 3, 4, 100};
        int t = 0;
        for (int x : u) t += x;
        System.out.println((double) t / u.length + " " + median(u));
        System.out.println(median(new int[]{8, 2, 6, 3}));
        System.out.println((3 + 6) / 2);
    }
}
```

```
22.0 3.0
4.5
4
```

The sorting determines the cost: O(n log n). The copy needs n cells of extra memory. There are also methods that find the median without sorting, in O(n) time on average, but they are outside the scope of this course.

Common mistake

A method that finds the median but sorts the array without copying it. Because the reference to the array is passed to the method, `Arrays.sort(arr)` sorts the caller's array too. After getting the median, the caller finds that the order of its own array has changed. The fix is to sort a copy taken inside the method with `arr.clone()` or `Arrays.copyOf`.

10.7.4 Bubble sort

Bubble sort compares neighboring elements and swaps those that are out of order. On each pass the largest of the remaining elements settles at the end, like a bubble rising to the surface of water. The Java version is two nested `for` loops:

```
import java.util.Arrays;

public class Bubble {
    public static void main(String[] args) {
        int[] a = {5, 1, 4, 2};
        int n = a.length;
        for (int pass = 0; pass < n - 1; pass++) {
          for (int j = 0; j < n - 1 - pass; j++) {
            if (a[j] > a[j + 1]) {
              int tmp = a[j];
              a[j] = a[j + 1];
              a[j + 1] = tmp;
            }
          }
        }
        System.out.println(Arrays.toString(a));
    }
}
```

```
[1, 2, 4, 5]
```

The outer loop counts the passes: n − 1 passes are enough for n elements, because once n − 1 elements are in place the last one is automatically in place. The bound of the inner loop is `j < n - 1 - pass`. This bound has two array-specific details. The first is the "`- 1`": since the body accesses `a[j + 1]`, `j` can be at most `n - 2`. If the bound were `j < n`, `a[n]` would be read on the last iteration and an out-of-bounds error would result. The second is the "`- pass`": since each pass puts one element at the end in its place, there is no need to look at that region again.

Pass	Steps in the inner loop	Array at the end of the pass	Comparisons	Swaps
1	5↔1, 5↔4, 5↔2	[1, 4, 2, 5]	3	3
2	1–4 in order, 4↔2	[1, 2, 4, 5]	2	1
3	1–2 in order	[1, 2, 4, 5]	1	0

Table 10.17: The passes of bubble sort on the array `{5, 1, 4, 2}`. The values come from the JDK 21 output of a version with a counter added.

The total number of comparisons is 3 + 2 + 1 = 6. In general, for n elements (n − 1) + (n − 2) + … + 1 = n(n − 1)/2 comparisons are made; this is $O(n^2)$. In the third pass there is no swap at all, because the array was already sorted after the second pass. This observation leads to an improvement: if no swap is made in a pass, the loop can stop early. This improvement is one of the exercises in the slides. The sort is done in place; the extra memory is only the variable `tmp`: O(1).

10.7.5 Insertion sort

Insertion sort grows a sorted part on the left of the array. On each iteration the element immediately to the right of the sorted part is taken as the **key**. The elements in the sorted part that are larger than the key are shifted one cell to the right, and the key is written into the gap that opens up. It resembles arranging playing cards in your hand.

```
import java.util.Arrays;

public class Insertion {
    public static void main(String[] args) {
        int[] a = {4, 3, 5, 1};
        for (int i = 1; i < a.length; i++) {
          int key = a[i];
          int j = i - 1;
          while (j >= 0 && a[j] > key) {
            a[j + 1] = a[j];
            j--;
          }
          a[j + 1] = key;
        }
        System.out.println(Arrays.toString(a));
    }
}
```

```
[1, 3, 4, 5]
```

i	key	Elements shifted right	Index where the key is written	Array at the end of the iteration
1	3	4	0	[3, 4, 5, 1]
2	5	—	2	[3, 4, 5, 1]
3	1	5, 4, 3	0	[1, 3, 4, 5]

Table 10.18: The iterations of insertion sort on the array `{4, 3, 5, 1}`.

The code has three points specific to arrays. First, the key is saved in a separate variable with `int key = a[i];`. This is necessary because the shifting will overwrite `a[i]`. The intermediate steps of the third iteration show this clearly. During the shifts the array goes through the states `[3, 4, 5, 5]`, `[3, 4, 4, 5]` and `[3, 3, 4, 5]`. During this time the value 1 lives only in `key`. Second, the check `j >= 0` in the `while` condition comes first. The `&&` operator is evaluated from left to right and does not look at the right side if the left side is false. When `j` is −1, `a[j]` is not read and an out-of-bounds error is avoided. If the order of the conditions were reversed (`a[j] > key && j >= 0`), the program would stop at `a[-1]` whenever the key is the smallest in the array. Third, when the loop ends, the key is written to `a[j + 1]`, because `j` stops one place to the left of the last element shifted.

In the worst case, when the array is in reverse order, every key shifts the whole sorted part and the cost is $O(n^2)$. If the array is already almost sorted, there is very little shifting and the cost approaches $O(n)$. This is why insertion sort is a good choice for small or nearly sorted arrays.

10.7.6 Binary search

Binary search works only on a sorted array. It looks at the element in the middle of the search range. If the target is larger than the middle element, the left half is eliminated; if smaller, the right half is. Because the range is halved at every step, an array of n elements needs at most about $\log_2 n + 1$ iterations. In the Java version the range is kept with two indices: `lo` (lower bound) and `hi` (upper bound).

```
public class Binary {
    public static void main(String[] args) {
        int[] a = {2, 5, 8, 12, 16, 23, 38, 56};
        int target = 23, lo = 0, hi = a.length - 1;
        int result = -1;
        while (lo <= hi) {
          int mid = lo + (hi - lo) / 2;
          if (a[mid] == target) { result = mid; break; }
          if (a[mid] < target) lo = mid + 1;
          else hi = mid - 1;
        }
        System.out.println(result);
    }
}
```

```
5
```

Iteration	`lo`	`hi`	`mid`	`a[mid]`	Decision
1	0	7	3	12	12 < 23 → `lo` = 4
2	4	7	5	23	found: `result` = 5

Table 10.19: Searching for 23 in the array `{2, 5, 8, 12, 16, 23, 38, 56}`.

23 is found in two iterations. Linear search would make six comparisons for the same target. In a small array the difference looks slight. In a sorted array of a million elements, however, binary search needs at most 20 iterations, while linear search on the same array may make up to a million comparisons.

The array-specific points are these. Initially `hi = a.length - 1`; `a.length` is not a valid index. The loop condition is `lo <= hi`: when `lo` and `hi` are equal, there is still one element in the range and it must be examined. `mid = lo + (hi - lo) / 2` is integer division and rounds down: 0 + (7 − 0) / 2 =

3. This form gives the same result as `(lo + hi) / 2`, but for very large indices it prevents the sum `lo + hi` from overflowing (Chapter 4.1). Finally, the ±1 in the updates `lo = mid + 1` and `hi = mid - 1` is essential. `mid` has already been examined. If the update is written as `lo = mid`, the range may fail to shrink in some cases and the loop runs forever.

10.7.7 Unsuccessful binary search

If the target is not in the array, the range eventually becomes empty: `lo` passes `hi`. At that moment `lo` shows the position where the target would go in the sorted array, that is, the insertion point. The method below returns this information by the same rule as `Arrays.binarySearch`:

```
import java.util.Arrays;

public class BinaryMiss {
    static int search(int[] a, int x) {
      int lo = 0, hi = a.length - 1;
      while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if (a[mid] == x) return mid;
        if (a[mid] < x) lo = mid + 1; else hi = mid - 1;
      }
      return -(lo + 1);
    }

    public static void main(String[] args) {
        System.out.println(search(new int[]{2, 4, 6, 8}, 5));
        System.out.println(Arrays.binarySearch(new int[]{2, 4, 6, 8}, 5));
        System.out.println(search(new int[]{2, 4, 6, 8}, 1) + " " + search(new int[]{2, 4,
6, 8}, 9));
    }
}
```

```
-3
-3
-1 -5
```

Iteration	lo	hi	mid	a[mid]	Decision
1	0	3	1	4	4 < 5 → `lo` = 2
2	2	3	2	6	6 > 5 → `hi` = 1
—	2	1			`lo > hi`: the loop ends

Table 10.20: Searching for 5 in the array `{2, 4, 6, 8}`.

When the loop ends, `lo` = 2 and the method returns −(2 + 1) = −3. The second line of the program shows that `Arrays.binarySearch` also returns −3 for the same array and the same target. A negative result means "not present". The insertion point is recovered from the result: −(−3) − 1 = 2. Indeed, 5 would go between 4 and 6, at position 2. The third line tests two edge cases. For 1, which is smaller than the smallest element, the insertion point is 0 and the return value is −1. For 9, which is larger than the largest, the insertion point is 4 and the return value is −5.

10.7.8 Basic complexities

The costs of the array operations seen in these notes follow directly from the structure of the array. Table 10.21 gives them together.

Operation	Time	Reason
Access by index	O(1)	The address is computed with a single multiplication and addition
Linear search	O(n)	In the worst case every element is examined
Full traversal	O(n)	Each element is visited once
Insert / delete in the middle	O(n)	The following elements are shifted one cell
Binary search (sorted array)	O(log n)	The range is halved at every step
Two nested full traversals	$O(n^2)$	For each element the whole array is examined

Table 10.21: Time complexities of basic array operations (n: number of elements).

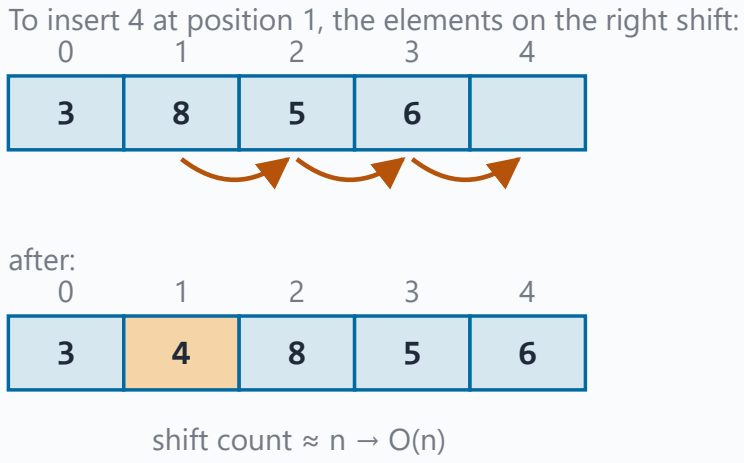


Figure 10.17: To insert 4 at position 1, 8, 5 and 6 each shift one cell to the right. The number of shifts is proportional to the number of elements: O(n).

Figure 10.17 makes the cost of inserting in the middle concrete. It is assumed that there is a free cell at the end of the array. Every element after the insertion point moves one place to the right; the shifting is done from right to left, otherwise the elements overwrite one another. Bubble sort and insertion sort use two nested loops, so they fall in the $O(n^2)$ row. `Arrays.sort`, with its O(n log n) cost, sits between these rows.

10.8 Two-dimensional arrays

The grades of the students in a class on three exams, the pixel colors of a picture, or the squares of a game board are arranged in rows and columns. For this kind of data, a structure with two indices is more natural: one selects the row and the other the column.

10.8.1 Declaring a matrix and accessing it

> **Definition 10.6**
>
> A **two-dimensional array** is an array whose elements are accessed with two indices. This structure of rows and columns is also called a **matrix**. In Java, a two-dimensional array is an **array of arrays**: each cell of the outer array holds the reference to a row array.

The type is written `int[][]`. In an initializer, each row is given in its own curly braces. An empty matrix is created with `new int[2][3]`: 2 rows, 3 columns in each row, all cells 0.

```
public class Matrix {
    public static void main(String[] args) {
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6}
        };
        int value = matrix[1][2]; // 6
        System.out.println(value + " " + matrix.length + " " + matrix[0].length);
        int[][] empty = new int[2][3];
        System.out.println(empty[1][2]);
    }
}
```

```
6 2 3
0
```

In an access, the first index selects the row and the second the column. `matrix[1][2]` is column 2 of row 1, that is, 6. The reading goes from left to right: first `matrix[1]` finds the row array, then the cell numbered 2 of that array is reached.

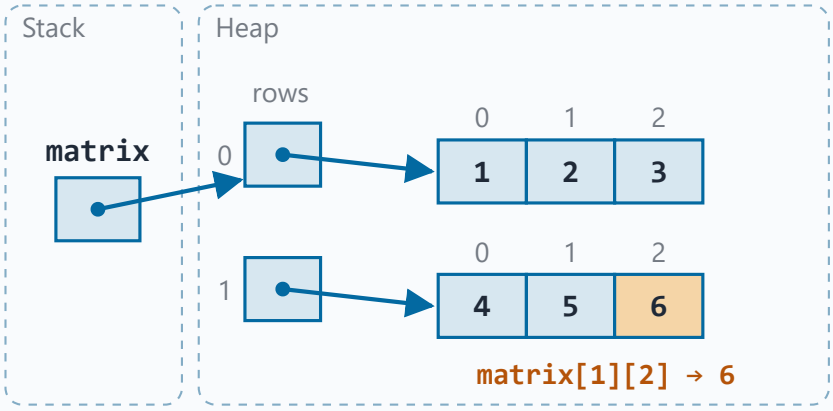


Figure 10.18: `int[][] matrix = {{1, 2, 3}, {4, 5, 6}};` in memory. The variable `matrix` points to the outer array of two cells. Each cell of the outer array points to a row array of three cells. The highlighted cell is `matrix[1][2]` = 6.

Figure 10.18 shows what the "array of arrays" idea looks like in memory. Two important consequences follow from this structure:

- `matrix.length` is the length of the outer array, that is, the **number of rows** (2). `matrix[0].length` is the length of row 0, that is, the **number of columns** in that row (3).
- `matrix[1]` on its own is a valid expression and is an `int[]`, a whole row. `Arrays.toString(matrix[1])` prints that row as `[4, 5, 6]`.

10.8.2 Jagged arrays

Because the rows are separate objects, they do not all have to be the same length. A two-dimensional array whose rows have different lengths is called a **jagged array**. To create one, only the number of rows is given in the `new` expression; the rows are created one by one afterward:

```
public class Jagged {
    public static void main(String[] args) {
        int[][] m = new int[3][];
        m[0] = new int[1];
        m[1] = new int[2];
        m[2] = new int[3];
        for (int i = 0; i < m.length; i++)
            System.out.print(m[i].length + " ");
        System.out.println();
    }
}
```

```
1 2 3
```

Line	`m[0]`	`m[1]`	`m[2]`	Output
`m = new int[3][];`	null	null	null	
`m[0] = new int[1];`	[0]	null	null	
`m[1] = new int[2];`	[0]	[0, 0]	null	
`m[2] = new int[3];`	[0]	[0, 0]	[0, 0, 0]	
loop, `i` = 0, 1, 2	same	same	same	`1 2 3`

Table 10.22: The trace table of the `Jagged` program.

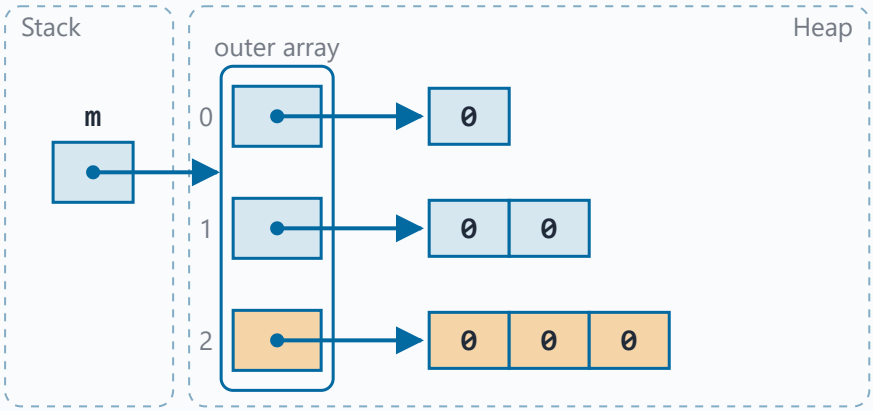


Figure 10.19: The jagged array after all three rows have been created. The cells of the outer array point to three separate row objects of lengths 1, 2 and 3.

`new int[3][]` creates only the outer array. Because the cells of the outer array hold references, their default values are `null`. If an expression such as `m[1].length` is run before the rows have been assigned, the program stops with a `NullPointerException`. The message from JDK 21 is:

```
> java JaggedNull
6
Exception in thread "main" java.lang.NullPointerException: Cannot read the array length
because "<local1>[1]" is null
        at JaggedNull.main(JaggedNull.java:7)
```

The message says “the array length cannot be read because `<local1>[1]` is null”. `<local1>` is the name of the variable `m` in the compiled code. If the source is compiled with the `javac -g` option, the message shows the real name of the variable:

```
> javac -g JaggedNull.java
> java JaggedNull
6
Exception in thread "main" java.lang.NullPointerException: Cannot read the array length
because "m[1]" is null
        at JaggedNull.main(JaggedNull.java:7)
```

Jagged arrays save memory for triangular data: Pascal's triangle, or half of a distance table.

10.8.3 The shallow-copy trap

In a two-dimensional array, `clone` copies only the outer array. The cells of the outer array hold the references of the rows, and it is these references that are copied. As a result, two separate outer arrays point to the **same** row objects.

```java
import java.util.Arrays;

public class Shallow {
    public static void main(String[] args) {
        int[][] a = {{1, 2}, {3, 4}};
        int[][] b = a.clone();
        b[0][0] = 99;
        b[1] = new int[]{7, 8};
        System.out.println(Arrays.deepToString(a));
        System.out.println(Arrays.deepToString(b));
    }
}
```

```
[[99, 2], [3, 4]]
[[99, 2], [7, 8]]
```

Line	`a`	`b`	Effect
`int[][] a = {{1, 2}, {3, 4}};`	[row 1, row 2]	—	Row 1 = [1, 2], row 2 = [3, 4]
`int[][] b = a.clone();`	[row 1, row 2]	[row 1, row 2]	New outer array, rows shared
`b[0][0] = 99;`	same	same	Row 1 = [99, 2]; `a[0][0]` is also 99
`b[1] = new int[]{7, 8};`	[row 1, row 2]	[row 1, row 3]	Only the cell of `b` changes

Table 10.23: The trace table of the `Shallow` program. Row 1 and row 2 are the two initial row objects; row 3 is created later.

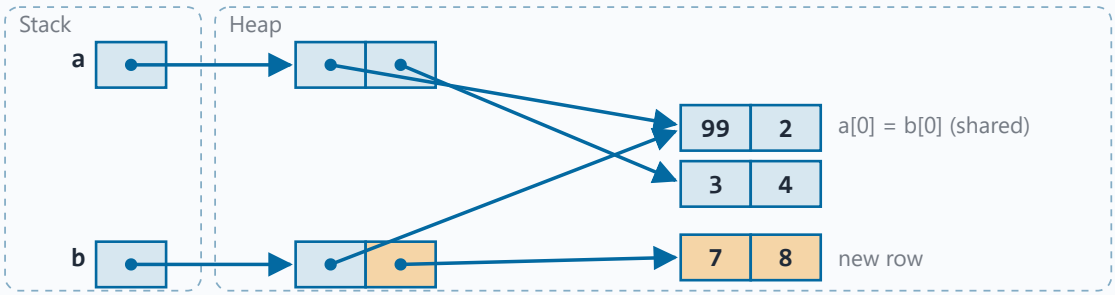


Figure 10.20: Memory after a new row is bound to `b[1]`. `a[0]` and `b[0]` point to the same row (99, 2). `a[1]` remains on the old row (3, 4), while `b[1]` points to the new row (7, 8).

The two lines giving different results are the heart of the trap. `b[0][0] = 99` writes into the shared row object, so it is seen from `a` too. `b[1] = new int[]{7, 8}`, on the other hand, writes not inside a row but into the cell of the outer array of `b`. That cell belongs only to `b` and `a` is not affected. For a true, that is **deep**, copy, each row is copied separately:

```
import java.util.Arrays;

public class Deep {
    public static void main(String[] args) {
        int[][] a = {{1, 2}, {3, 4}};
        int[][] b = new int[a.length][];
        for (int i = 0; i < a.length; i++)
            b[i] = a[i].clone();
        b[0][0] = 99;
        System.out.println(Arrays.deepToString(a));
        System.out.println(Arrays.deepToString(b));
    }
}
```

```
[[1, 2], [3, 4]]
[[99, 2], [3, 4]]
```

This time `b[0][0] = 99` changes only the row that belongs to `b`; `a` stays `[[1, 2], [3, 4]]`. The `b[i] = a[i].clone()` in the loop creates a separate object for each row.

10.8.4 Traversing a matrix row by row

To traverse all the cells of a matrix, two nested loops are used. The outer loop goes through the rows, and the inner loop goes through the columns of that row:

```
public class Traverse {
    public static void main(String[] args) {
        int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
        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();
        }
    }
}
```

```
1 2 3
4 5 6
```

The bound of the inner loop is `matrix[i].length`; that is, each row is traversed up to its own length. This form also works correctly for jagged arrays. The cells are printed on the same line with `print`, separated by spaces. When a row is finished, the inner loop is left and `println()` moves to the next line.

Step	`i`	`j`	`matrix[i][j]`	Line on the console
1	0	0	1	`1`
2	0	1	2	`1 2`
3	0	2	3	`1 2 3` ⏎
4	1	0	4	`4`
5	1	1	5	`4 5`
6	1	2	6	`4 5 6` ⏎

Table 10.24: The order in which cells are visited in the `Traverse` program.

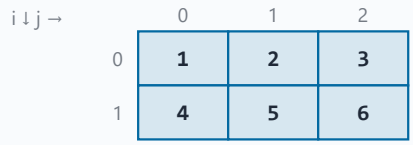


Figure 10.21: A 2 × 3 matrix. `i` selects the row going down and `j` the column going right. In row-by-row traversal, the three cells of row 0 are visited first, then the three cells of row 1.

In total 2 × 3 = 6 cells are traversed. In general, for a matrix with r rows and c columns, the cost is O(r · c). For a square n × n matrix this is $O(n^2)$. Operations such as a row sum, a column sum or finding the largest cell are all built on this skeleton. For the sum of each row, for example, `int sum = 0;` is written before the inner loop and the sum is printed when the inner loop ends.

10.8.5 Matrix transpose

The **transpose** of a matrix is the matrix in which rows become columns and columns become rows. The transpose of an r × c matrix is c × r, and it is filled by the rule `t[j][i] = matrix[i][j]`.

```
import java.util.Arrays;

public class Transpose {
    public static void main(String[] args) {
        int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
        int rows = matrix.length;
        int cols = matrix[0].length;
        int[][] t = new int[cols][rows];
        for (int i = 0; i < rows; i++)
            for (int j = 0; j < cols; j++)
                t[j][i] = matrix[i][j];
        System.out.println(Arrays.deepToString(t));
    }
}
```

```
[[1, 4], [2, 5], [3, 6]]
```

`i`	`j`	Assignment	`t` (at the end of the step)
0	0	`t[0][0] = matrix[0][0]` = 1	[[1, 0], [0, 0], [0, 0]]
0	1	`t[1][0] = matrix[0][1]` = 2	[[1, 0], [2, 0], [0, 0]]
0	2	`t[2][0] = matrix[0][2]` = 3	[[1, 0], [2, 0], [3, 0]]
1	0	`t[0][1] = matrix[1][0]` = 4	[[1, 4], [2, 0], [3, 0]]
1	1	`t[1][1] = matrix[1][1]` = 5	[[1, 4], [2, 5], [3, 0]]
1	2	`t[2][1] = matrix[1][2]` = 6	[[1, 4], [2, 5], [3, 6]]

Table 10.25: The trace table of the `Transpose` program.



Figure 10.22: The 2 × 3 `matrix` and its 3 × 2 transpose `t`. Row 0 of the matrix (1, 2, 3) has become column 0 of `t`.

In the code the dimensions are read first: `rows = matrix.length` (2) and `cols = matrix[0].length` (3). The target array is created with the dimensions swapped: `new int[cols][rows]`, that is, 3 × 2. Taking the number of columns from `matrix[0].length` assumes that all the rows are the same length; the transpose is meaningful only for regular matrices anyway. Each cell is copied once: O(r · c) time, and r · c cells of extra memory for the target matrix.

10.8.6 Matrix multiplication

The product of two matrices C = A × B is defined as follows: the cell of C in row `i` and column `j` is the sum of the products of the corresponding elements of row `i` of A and column `j` of B. The number of columns of A must equal the number of rows of B.

> C[i][j] = A[i][0] · B[0][j] + A[i][1] · B[1][j] + … + A[i][m−1] · B[m−1][j]

```
import java.util.Arrays;

public class MatMul {
    public static void main(String[] args) {
        int[][] A = {{1, 2}, {3, 4}};
        int[][] B = {{5, 6}, {7, 8}};
        int[][] C = new int[2][2];
        for (int i = 0; i < 2; i++)
          for (int j = 0; j < 2; j++)
            for (int k = 0; k < 2; k++)
              C[i][j] += A[i][k] * B[k][j];
        System.out.println(Arrays.deepToString(C));
    }
}
```

```
[[19, 22], [43, 50]]
```

Example 10.3

For A = [[1, 2], [3, 4]] and B = [[5, 6], [7, 8]], let us compute the four cells of C step by step. Each cell is the sum of two products, because A has 2 columns.

- C[0][0] = A[0][0] · B[0][0] + A[0][1] · B[1][0] = 1 · 5 + 2 · 7 = 5 + 14 = 19
- C[0][1] = A[0][0] · B[0][1] + A[0][1] · B[1][1] = 1 · 6 + 2 · 8 = 6 + 16 = 22
- C[1][0] = A[1][0] · B[0][0] + A[1][1] · B[1][0] = 3 · 5 + 4 · 7 = 15 + 28 = 43
- C[1][1] = A[1][0] · B[0][1] + A[1][1] · B[1][1] = 3 · 6 + 4 · 8 = 18 + 32 = 50

The result is [[19, 22], [43, 50]], the same as the JDK 21 output.

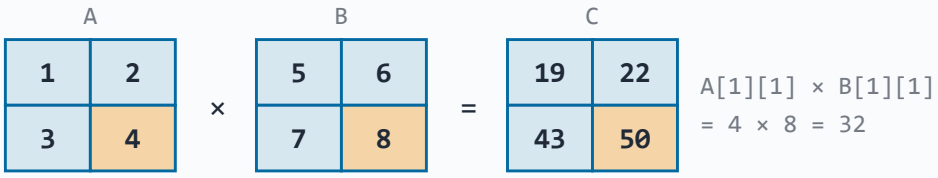


Figure 10.23: The last step of the last cell: C[1][1] += A[1][1] × B[1][1] = 4 × 8 = 32, so 18 + 32 = 50. Row 1 of A and column 1 of B have been used.

The code has three nested loops. `i` and `j` select the cell to be computed; `k` runs over the terms of the sum. The statement `C[i][j] += A[i][k] * B[k][j]` adds each term to the cell. Because `C` was created with `new int[2][2]`, its cells start at 0; there is no need to zero them separately. For square n × n matrices, n multiplications are made for each cell and n^3 multiplications for the n^2 cells in total: $O(n^3)$. For 100 × 100 matrices this is a million multiplications.

10.9 Array or list?

The length of an array is fixed. If the number of elements grows and shrinks while the program runs, this limit is annoying: with every insertion a new, larger array has to be created and the old elements copied into it. The `ArrayList` class in Java's collections library does this job by itself.

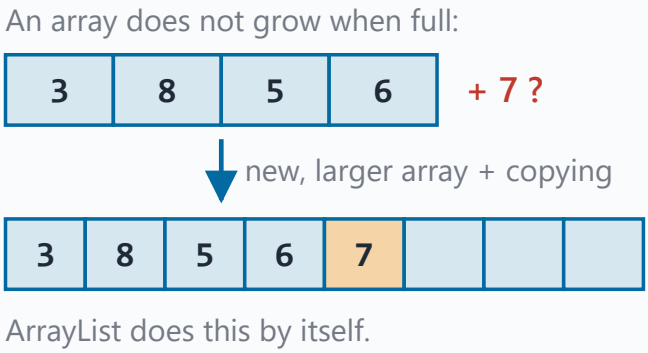


Figure 10.24: 7 cannot be added to a full array, because the array does not grow. A new, larger array is created, the elements are copied and 7 is written into an empty cell. `ArrayList` performs these steps by itself.

```
import java.util.ArrayList;

public class ListDemo {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<>();
        list.add(3);
        list.add(8);
        list.add(5);
        list.add(7);
        System.out.println(list + " " + list.size() + " " + list.get(1));
    }
}
```

```
[3, 8, 5, 7] 4 8
```

Internally, `ArrayList` still uses an array. When the array fills up, it creates a larger one and moves the elements there (Figure 10.24). The programmer only calls methods such as `add`, `get` and `size`. For access, `list.get(1)` is written instead of square brackets, and `list.size()` instead of `length` for the length.

This convenience has a price. An `int[]` stores the numbers directly in its cells. An `ArrayList<Integer>`, however, can hold only objects; each number is put inside an `Integer` object, and the list holds the references to these objects. This means more memory and somewhat slower access. A rough rule for choosing can be given:

- If the number of elements is known in advance and fixed, for example the days of the week, the cells of a matrix or a table of counters, an array is sufficient and fast.
- If the number of elements will change while the program runs, for example records entered by the user, a collection such as `ArrayList` is more convenient.

The final decision is made according to how it will be used, the readability of the code and, if necessary, measured performance. The traversal, search and sorting methods learned in these notes apply to lists in the same way.

10.10 Common mistakes

Most mistakes with arrays fall into four groups. This section gives each with an example that was actually run.



Figure 10.25: Two common mistakes. Left: one iteration too many, where `i <= a.length` accesses the nonexistent `a[5]` on the last iteration. Right: a reference copy, where after `b = a`, `b[0] = 9` changes the single shared array and `a[0]` becomes 9 too.

Common mistake

One iteration too many. The condition `i <= a.length` accesses `a[length]` on the last iteration, and an `ArrayIndexOutOfBoundsException` is thrown. The correct form is `i < a.length`. The same mistake has a silent sibling: writing `i < a.length - 1` skips the last element and gives no message.

Common mistake

The first element of an empty array. `int min = a[0];` stops with the message "Index 0 out of bounds for length 0" on an empty array. Before computing the minimum, the maximum or the median, the case `a.length == 0` is handled separately.

Common mistake

A reference copy. `int[] b = a;` is not a new array; `b[0] = 9` changes `a` too. For an independent copy, use `a.clone()` or `Arrays.copyOf`. For two-dimensional arrays these are not enough either; each row is copied separately.

Common mistake

A fixed number of columns in a jagged matrix. Using `matrix[0].length` for every row in the inner loop assumes that all the rows are as long as the first. In the following program the second row has only one element:

```
public class JaggedError {
    public static void main(String[] args) {
        int[][] m = {{1, 2, 3}, {4}};
        int sum = 0;
        for (int i = 0; i < m.length; i++)
            for (int j = 0; j < m[0].length; j++)
                sum += m[i][j];
        System.out.println(sum);
    }
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 1 out of
bounds for length 1
        at JaggedError.main(JaggedError.java:7)
```

When `i` = 1, `j` becomes 1 and cell number 1 is looked up in a row of length 1. When the inner loop bound is changed to `m[i].length`, the program prints 10.

What these four mistakes have in common is that the compiler catches none of them. All of them show up either as an exception at run time or as a wrong result with no message at all. So when writing array code, three boundary cases are always tried: an empty array, an array with a single element, and an array in which the target is in the first or last position.

10.11 Summary

- An **array** is an object that holds a fixed number of elements of the same type in consecutive cells. Indices run from 0 to `length - 1`; `length` is a field and does not change after creation.
- An array variable lives on the stack and holds the **reference** to the array object on the heap. Cells created with `new` receive the default value: 0 for numbers, `false` for `boolean` and `null` for objects. The initializer `{…}` takes the length from the number of values.
- Reading and writing `a[i]` are done in O(1) time, because the address is computed with the formula *start + i × cell size*. An invalid index stops the program with an `ArrayIndexOutOfBoundsException`; the most common cause is a loop condition written with `<=`.
- The classic `for` is used when an index is needed, and the enhanced `for` when you only read. The loop variable in the enhanced `for` is a copy of the element; assigning to it does not change the array.
- The sum and average are found with an accumulator, and the minimum and maximum with candidates initialized to the first element of the array. Reversing is done in place with two pointers. In frequency counting the value is used as the index. The sieve of Eratosthenes eliminates the non-primes with a `boolean` array.

- `b = a` copies only the reference. `clone`, `Arrays.copyOf` and `System.arraycopy` copy the elements, but make shallow copies. `==` and `equals` compare references, while `Arrays.equals` compares contents. An array passed to a method is the same object as the caller's array.
- Linear search takes O(n) time, and binary search on a sorted array O(log n). `Arrays.sort` sorts the array in place; `Arrays.binarySearch` returns `-(insertion point) - 1` if it does not find the target. Bubble sort and insertion sort are $O(n^2)$.
- `int[][]` is an array of arrays. `matrix.length` is the number of rows, and `matrix[i].length` is the number of columns of that row. Rows can have different lengths (a jagged array). `clone` copies only the outer array. Traversal is done with two nested loops, and matrix multiplication with three.

10.12 Key terms

array element cell index length reference heap stack default value array initializer null reference out of bounds exception traversal enhanced for loop reversing pointer swap frequency Sieve of Eratosthenes copy shallow copy linear search binary search median bubble sort insertion sort time complexity two-dimensional array multidimensional array matrix jagged array

---

10.13 Test Yourself

1. What is the output of the following program?

```
import java.util.Arrays;

public class TY1 {
    public static void main(String[] args) {
        int[] a = new int[4];
        a[a.length - 1] = a.length;
        a[a[3] - 3] = 9;
        System.out.println(Arrays.toString(a));
    }
}
```

Answer

```
[0, 9, 0, 4]
```

The array starts with four zeros. `a[a.length - 1] = a.length` writes 4 into the last cell, `a[3]`. In the next line the inner expression is computed first: `a[3] - 3` = 4 − 3 = 1. So 9 is written into `a[1]`.

2. In the following program, what is `a` after the method `zeroOut` has been called?

```
import java.util.Arrays;

public class TY2 {
    static void zeroOut(int[] d) {
        d[0] = 0;
        d = new int[]{5, 5, 5};
        d[1] = 0;
    }

    public static void main(String[] args) {
        int[] a = {1, 2, 3};
        zeroOut(a);
        System.out.println(Arrays.toString(a));
    }
}
```

Answer

```
[0, 2, 3]
```

The reference of `a` goes to the method. `d[0] = 0` writes into the shared array, and `a[0]` becomes 0. `d = new int[]{5, 5, 5}`, however, only binds the method's local variable to a new array. The following `d[1] = 0` changes this new array; `a` is not affected.

3. What is the output of the following program? Why is each of the three values what it is?

```
import java.util.Arrays;

public class TY3 {
    public static void main(String[] args) {
        int[] p = {4, 5};
        int[] q = p.clone();
        int[] r = p;
        System.out.println((p == q) + " " + (p == r) + " " + Arrays.equals(p, q));
    }
}
```

Answer

```
false true true
```

`p == q` is false, because `clone` creates a new object. `p == r` is true, because `r = p` copies the same reference. `Arrays.equals(p, q)` is true, because the two arrays have the same length and the same elements.

4. `Arrays.binarySearch` looks for 1 first in an unsorted array and then in a sorted one. What is the output, and why is the first result wrong?

```
import java.util.Arrays;

public class TY4 {
    public static void main(String[] args) {
        int[] a = {5, 1, 9, 3, 7};
        System.out.println(Arrays.binarySearch(a, 1));
        Arrays.sort(a);
        System.out.println(Arrays.binarySearch(a, 1));
    }
}
```

Answer

```
-1
0
```

In the first search the range is 0…4 and the middle element is `a[2]` = 9. Since 1 < 9, the right half is eliminated. Then `a[0]` = 5 is looked at; since 1 < 5, the search goes left again and the range becomes empty. The method returns −1, although 1 is in the array, at position 1. Because the method assumes the array is sorted, it has eliminated the wrong half. After sorting, the array is `[1, 3, 5, 7, 9]` and 1 is correctly found at position 0.

5. The following program creates a triangular jagged array. What are `t[3].length` and the total number of cells?

```
public class TY5 {
    public static void main(String[] args) {
        int[][] t = new int[4][];
        int cells = 0;
        for (int i = 0; i < t.length; i++) {
            t[i] = new int[i + 1];
            cells += t[i].length;
        }
        System.out.println(t[3].length + " " + cells);
    }
}
```

Answer

```
4 10
```

Row number `i` has `i + 1` cells: 1, 2, 3 and 4. The length of the last row is 4, and the total number of cells is 1 + 2 + 3 + 4 = 10. A regular 4 × 4 matrix would need 16 cells.

6. The reverse-sorted array `{4, 3, 2, 1}` is sorted with bubble sort. How many comparisons and

how many swaps are made?

```
import java.util.Arrays;

public class TY6 {
    public static void main(String[] args) {
        int[] a = {4, 3, 2, 1};
        int comps = 0, swaps = 0;
        for (int pass = 0; pass < a.length - 1; pass++)
            for (int j = 0; j < a.length - 1 - pass; j++) {
                comps++;
                if (a[j] > a[j + 1]) {
                    int tmp = a[j]; a[j] = a[j + 1]; a[j + 1] = tmp;
                    swaps++;
                }
            }
        System.out.println(Arrays.toString(a) + " " + comps + " " + swaps);
    }
}
```

Answer

```
[1, 2, 3, 4] 6 6
```

The number of comparisons does not depend on the contents of the array: 3 + 2 + 1 = 6. In a reverse-sorted array, every comparison finds an out-of-order pair and ends in a swap; the number of swaps is also 6. This is the worst case for n = 4: $n(n-1)/2 = 6$.

7. The following program looks for the most frequent digit in an array. What is the output? Why does the `counts` array have 10 cells?

```
public class TY7 {
    public static void main(String[] args) {
        int[] a = {2, 7, 7, 3, 7};
        int[] counts = new int[10];
        for (int x : a) counts[x]++;
        int mostCommon = 0;
        for (int d = 1; d < counts.length; d++)
            if (counts[d] > counts[mostCommon]) mostCommon = d;
        System.out.println(mostCommon + " " + counts[mostCommon]);
    }
}
```

Answer

```
7 3
```

The values are used as indices: `counts[x]++`. Because digits are between 0 and 9, 10 cells are needed. At the end of counting, `counts[2]` = 1, `counts[3]` = 1 and `counts[7]` = 3. The second loop looks for the index of the largest counter; since 7 occurs three times, the result is `7 3`.

8. The following program fills a 3 × 3 matrix. What are `m[2][1]` and `m[1][2]`?

```
public class TY8 {
    public static void main(String[] args) {
        int[][] m = new int[3][3];
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++)
                m[i][j] = i * 3 + j;
        System.out.println(m[2][1] + " " + m[1][2]);
    }
}
```

Answer

```
7 5
```

Each cell is given `i * 3 + j`; the cells end up numbered row by row from 0 to 8. `m[2][1]` = 2 · 3 + 1 = 7 and `m[1][2]` = 1 · 3 + 2 = 5. Because the first index is the row and the second the column, the two expressions refer to different cells.

9. To print an array of length `n` from the end to the start, a programmer writes the loop `for (int i`

`= a.length; i >= 0; i--)`. What error occurs, and how is the loop corrected?

Answer

In the first iteration `i = a.length`, and an attempt is made to read `a[a.length]`. Run with a three-element array, the program stops without printing anything:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out
of bounds for length 3
        at TY9.main(TY9.java:5)
```

Because the last valid index is `a.length - 1`, the correct start is `int i = a.length - 1`. The condition `i >= 0` is right, because cell 0 must also be printed.

10.14 Further reading

- Oracle, *The Java Tutorials*, "Arrays": explains array declaration, creation, multidimensional arrays, copying and the `java.util.Arrays` methods with short, runnable examples.
- R. Sedgewick and K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), 1.4 Arrays: includes array applications such as shuffling a deck, the coupon collector problem and the sieve of Eratosthenes; the book's website (introcs.cs.princeton.edu) has program code and exercises.
- A. B. Downey and C. Mayfield, *Think Java* (2nd ed., O'Reilly, 2019), Chapter 7 "Arrays and References": explains the concept of a reference, passing arrays to methods and traversal patterns in plain language for beginners; a free online version is available.
- Oracle, *The Java Language Specification, Java SE 21 Edition*, Chapter 10 "Arrays": gives the exact definition of array types, initializers, default values and the behavior of `clone`.

10.15 References

1. P. Deitel, H. Deitel, *Java: How to Program, Early Objects*, 11th ed., Pearson, 2018.
2. Y. D. Liang, *Introduction to Java Programming and Data Structures*, 13th ed., Pearson, 2024.
3. Oracle, *The Java Tutorials*: Arrays.
4. Oracle, *Java SE 21 API*: java.util.Arrays; *The Java Language Specification, Java SE 21 Edition*: Chapter 10. Arrays.
5. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 1.4 Arrays.
6. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
7. A. B. Downey, C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019 (free online version).

Strings

Most of the data programs process is text: student names, email addresses, sentences typed at the keyboard, reports printed to the screen. In Java, text is represented by objects of the `String` class. These notes explain how strings are held in memory, why they cannot be changed, and their most frequently used methods. Reaching characters by index, taking substrings, searching, splitting and replacing are treated one by one. Comparison of strings follows: the difference between `==` and `equals`, lexicographic order with `compareTo`, and the traps caused by Turkish letters. Reading text from the keyboard, efficient concatenation with `StringBuilder` and formatted output with `printf` are also among the topics. The applications in the last section (reversing, palindromes, counting vowels and words, letter frequency, the Caesar cipher) bring all these tools together. Every program was compiled and run with JDK 21; the outputs given are the real outputs of the programs.

LEARNING OUTCOMES

Representation
Explain the difference between `String` and `char`, that a string variable holds a reference and that strings are immutable; tell the empty string from `null`.

Processing
Reach characters with an index and `charAt`; use methods such as `length`, `indexOf`, `substring`, `split` and `replace` with correct bounds; scan a string with a loop.

Comparison
Tell content comparison (`equals`) from reference comparison (`==`); compute the result of `compareTo`; explain the effect of the locale on Turkish letters.

Input and output
Read words and lines with `Scanner` and trace the buffer behavior; concatenate efficiently with `StringBuilder`; produce formatted text with `printf`.

11.1 What is a string?

Most of the programs written so far worked with numbers. A number is held in an `int` or `double` variable and goes into arithmetic operations. Text is a different structure: an ordered sequence of letters, digits, spaces and punctuation marks. This section introduces the two building blocks of text in Java, the types `char` and `String`.

11.1.1 Characters and strings

> **Definition 11.1**
>
> A **character** is a single letter, digit, space or mark of text. In Java a character is held in a variable of type `char` and is written in single quotes: `'T'`, `'7'`, `' '`. A **string** is an ordered sequence of zero or more characters. In Java, strings are objects of the `String` class and are written in double quotes: `"Tarkan"`, `"7"`, `""`.

The kind of quote matters. `'T'` is a single character and its type is `char`. `"T"` is a string of length 1 and its type is `String`. The two look the same on the screen but are different types for Java. This is why a string cannot be assigned to a `char` variable:

```
public class WrongType {
    public static void main(String[] args) {
        char c = "T";
        System.out.println(c);
    }
}
```

```
WrongType.java:3: error: incompatible types: String cannot be converted to char
        char c = "T";
                 ^
1 error
```

The message says that the type `String` cannot be converted to the type `char`. The correct form is `char c = 'T';`.

The `String` class is in the `java.lang` package. This package is added to every Java program automatically; the `import` line needed for `Scanner` is not needed for `String`. A piece of text written directly in the program, such as `"Tarkan"`, is called a **string literal**.

11.1.2 A string is an object

Primitive types such as `char`, `int` and `double` hold their values directly inside the variable. `String`, however, is not a primitive type; it is a class. A `String` variable holds not the text itself but the location in memory of the object that carries the text. This location information is called a **reference**. Figure 11.1 compares how the two types look in memory.

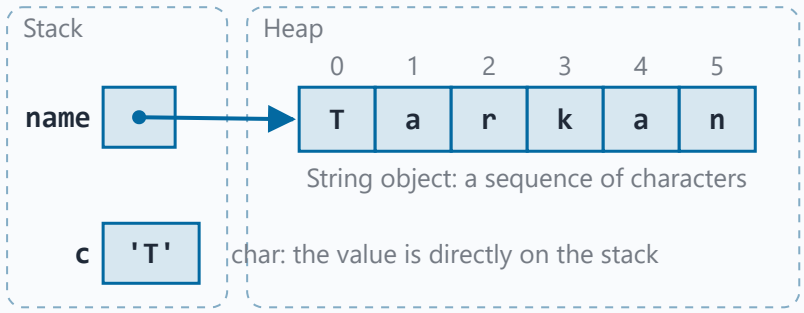


Figure 11.1: Memory after the declarations `String name = "Tarkan";` and `char c = 'T';`. The variable `name` holds an arrow (a reference) pointing to the six-character object on the heap. The variable `c` carries the value `'T'` directly inside itself.

The names of the two memory regions in the figure appear in all the later drawings. The **stack** is the region where the local variables of methods are kept. The **heap** is the region where objects created with `new`, and strings, live. A `String` variable on the stack points to the object on the heap with an arrow. Several variables can point to the same object. This sharing is decisive when strings are compared.

Definition 11.2

An **object** is a piece of memory produced from a class, carrying its own data and methods. A **reference** is a value that shows where an object is in memory. In the declaration `String s = "Ada";`, the variable `s` is a reference; `"Ada"` is the object on the heap.

One consequence of being an object is methods. Operations belonging to a string are called with the dot: `name.length()` gives the length of the string and `name.charAt(0)` its first character. Primitive types have no methods; a notation such as `c.length()` does not compile.

11.1.3 Creating strings, the empty string and null

A string variable can be given an initial value in four different ways:

```
String name = "Tarkan";
String empty = "";
String copy = new String(name);
String none = null;
```

The first line is the most common form: `name` points to the constant `"Tarkan"`. The `""` in the second line is the **empty string**, which contains no characters. The third line creates a new object with `new` and copies the contents of `name` into it. The `null` in the fourth line means that the variable points to no object at all. Figure 11.2 shows the state of the four variables in memory.

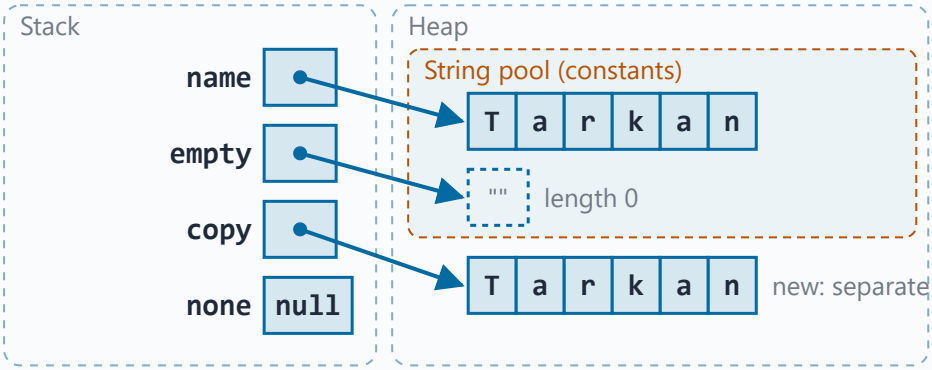


Figure 11.2: Four string variables. `name` and `empty` point to constants in the string pool. `copy` points to a separate object created with `new`; even though its contents are the same, the object is different. `none` points to no object.

The **string pool** in the figure is a special region inside the heap where a single copy of each constant is kept. The rule here is short: a string created with a literal comes from the pool, while `new` always creates a new object. In practice there is almost never a need to write `new String(...)`; the string literal is preferred.

The empty string and `null` are often confused. The empty string is a real object; its length is 0 and all its methods can be called. `null` is not an object; it is the state of a variable that says "I do not point to anything yet". If a method is called through a `null` variable, the program stops with a `NullPointerException`. The program below prints the four variables and then calls `none.length()`:

```
public class Creation {
    public static void main(String[] args) {
        String name = "Tarkan";
        String empty = "";
        String copy = new String(name);
        String none = null;
        char c = 'T';
        System.out.println(name + " " + name.length());
        System.out.println("[" + empty + "] " + empty.length());
        System.out.println(copy + " " + copy.length());
        System.out.println(none);
        System.out.println(c);
        System.out.println(none.length());
    }
}
```

```
Tarkan 6
[] 0
Tarkan 6
null
T
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()"
because "none" is null
```

In the output you can see that the empty string shows nothing between the square brackets and that its length is 0. `System.out.println(none)` does not cause an error; it prints `null`. The error occurs only when a method is called through `null`. The JDK 21 message also states the cause of the problem: because `none` is null, `String.length()` cannot be called.

Common mistake

Giving `null` to a string variable whose value is not known yet and then calling a method on it. If the variable will be filled in a loop or a condition, making its initial value `""` is often safer. If `null` really means “no value”, a check `s != null` is made before the method call.

11.1.4 Concatenation with +

Joining two strings end to end is called **string concatenation**. In Java the `+` operator is used for this. The same operator does addition on numbers. Which one happens is decided by the types of the operands: if at least one of the two operands is a `String`, `+` is concatenation; if both are numbers, it is addition.

```
public class Concatenation {
    public static void main(String[] args) {
        String num1 = "12";
        String num2 = "34";
        String total = num1 + num2;
        System.out.println(total);
        int a = 12, b = 34;
        System.out.println(a + b);
        System.out.println(num1 + 3 + 4);
        System.out.println(3 + 4 + num1);
        System.out.println(num1 + (3 + 4));
        System.out.println("Total: " + a + b);
        System.out.println("Total: " + (a + b));
        System.out.println("x" + null + 'c' + 2.5 + true);
        System.out.println("Merhaba".concat(" dünya"));
    }
}
```

```
1234
46
1234
712
127
Total: 1234
Total: 46
xnullc2.5true
Merhaba dünya
```

The program works line by line as follows. `num1 + num2` concatenates two strings; even though the variable is named “total”, the result is `"1234"`. `a + b` adds two `int`s: 46. The next two lines show the effect of the order of evaluation. `+` is evaluated from left to right:

- `num1 + 3 + 4`: first *"12" + 3* is computed. Because one operand is a string, 3 is converted to a string: `"123"`. Then *"123" + 4* = `"1234"`.
- `3 + 4 + num1`: first *3 + 4* is computed. Both are numbers: 7. Then *7 + "12"* = `"712"`.
- `num1 + (3 + 4)`: the parentheses are computed first: 7. Then *"12" + 7* = `"127"`.

The line `"Total: " + a + b` falls into the same trap: because of the string on the left, both numbers are appended to the text one by one and “Total: 1234” is printed. To print the sum, parentheses are needed: `"Total: " + (a + b)`. `null`, `char`, `double` and `boolean` values can also be added to a string; each is converted to its text form. The `concat` method also joins two strings, but it accepts only strings.

Example 11.1

With `int x = 5, y = 2;`, what are the values of the expressions `"x+y=" + x + y` and `x + y + "=x+y"`?

Solution. In the first expression the leftmost operand is a string. *"x+y=" + 5* = `"x+y=5"`, then *"x+y=5" + 2* = `"x+y=52"`. In the second expression the first two operands are numbers: *5 + 2 = 7*. Then *7 + "=x+y"* = `"7=x+y"`. The same three operands give a different result when their order changes.

11.1.5 A char is a number

Computers store characters as numbers. In the Unicode standard every character has a code number: `'a'` is 97, `'b'` is 98, `'A'` is 65 and `'0'` is 48. In Java, `char` is really a 16-bit unsigned integer type. So a `char` can take part in arithmetic. The type of the result follows a rule: when a `char` is added to an integer, or two `char`s are added, the result is `int`.

```
public class CharArithmetic {
    public static void main(String[] args) {
        char k = 'a';
        System.out.println(k + 1);
        System.out.println((char) (k + 1));
        System.out.println("" + k + 1);
        System.out.println('a' + 'b');
        System.out.println("" + 'a' + 'b');
        k++;
        System.out.println(k);
    }
}
```

```
98
b
a1
195
ab
b
```

`k + 1` is an `int`: *97 + 1 = 98*. `(char) (k + 1)` converts this number back to a character: `b`. In the expression `"" + k + 1`, the empty string on the far left turns the whole expression into concatenation: first `"a"`, then `"a1"`. `'a' + 'b'`, however, is the sum of two numbers: *97 + 98 = 195*. To print two characters side by side, `""` is again put at the start. `k++` increases the `char` variable in place and its type does not change: `b`.

That characters are numbers is useful in most string operations. The difference of two lowercase letters gives their distance in the alphabet: *'k' − 'a' = 107 − 97 = 10*. Subtracting `'0'` from a digit character gives the numeric value of the digit: *'7' − '0' = 55 − 48 = 7*. The letter-frequency and Caesar cipher applications rest on these two ideas.

```
public class CharNumber {
    public static void main(String[] args) {
        System.out.println((int) 'a' + " " + (int) 'k' + " " + (int) 'v' + " " + (int) 'z'
+ " " + (int) 'A' + " " + (int) '0');
        System.out.println('k' - 'a');
        System.out.println((char) ('a' + 10));
        System.out.println((int) 'ç' + " " + ('ç' - 'a') + " " + (int) 'ı' + " " + (int)
'İ' + " " + (int) 'ş');
        System.out.println('7' - '0');
    }
}
```

```
97 107 118 122 65 48
10
k
231 134 305 304 351
7
```

The last lines also show the codes of Turkish letters: `'ç'` is 231, `'ı'` is 305, `'İ'` is 304 and `'ş'` is 351. These codes lie outside the range 97–122 of the English alphabet. Because *'ç' − 'a' = 134*, there is no room for `'ç'` in a 26-cell letter table running from `'a'` to `'z'`.

11.2 Immutability

The most important property of strings is that their contents do not change after they are created. At first this property seems strange: when you write `s = s + "!"`, it looks as if `s` has changed. In reality, what changes is not the object but the place the variable points to.

11.2.1 A new object, the same old object

Definition 11.3

An object whose contents cannot be changed after it is created is called an **immutable** object. `String` is an immutable class: none of its methods changes the characters of the object. Every operation that "changes" a string (concatenation, converting to uppercase, taking a substring, replacing) produces a new `String` object and returns it. This property is called **immutability**.

Figure 11.3 shows the simplest example. After the line `String s = "Merhaba";`, `s` points to the seven-character object. The line `s = s + "!";` does not add an exclamation mark to this object. First `s + "!"` is computed and a new eight-character object is created. Then the assignment moves the arrow of `s` to this new object. The old `"Merhaba"` object stays as it was.

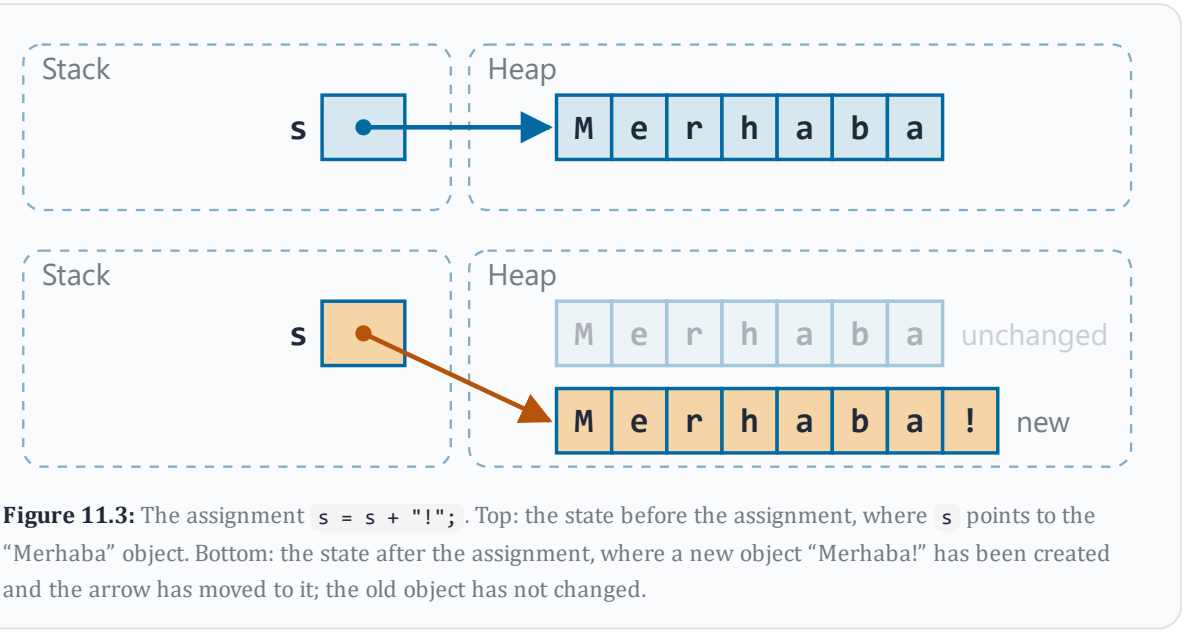


Figure 11.3: The assignment `s = s + "!";`. Top: the state before the assignment, where `s` points to the "Merhaba" object. Bottom: the state after the assignment, where a new object "Merhaba!" has been created and the arrow has moved to it; the old object has not changed.

This can also be seen with a program. If a second variable (`old`) is kept pointing to the old object before the assignment, it becomes clear after the assignment that the two variables point to different objects:

```
public class Exclaim {
    public static void main(String[] args) {
        String s = "Merhaba";
        String old = s;
        s = s + "!";
        System.out.println(s);
        System.out.println(old);
        System.out.println(s == old);
    }
}
```

```
Merhaba!
Merhaba
false
```

`old` still points to "Merhaba". The comparison `s == old` asks whether the two variables point to the same object, and it gives `false`: there are now two separate objects. Objects that no variable points to become unnecessary in time. In Java the **garbage collector** cleans up these objects automatically; the programmer does not have to give memory back by hand.

11.2.2 Tracing the toUpperCase example step by step

The practical consequence of immutability is this: if the result of a string method is wanted, it must be assigned to a variable. The five lines below show this point.

```
String lang = "java";
lang.toUpperCase();
System.out.println(lang); // java
lang = lang.toUpperCase();
System.out.println(lang); // JAVA
```

```
java
JAVA
```

The second line calls `lang.toUpperCase()` but puts the returned value nowhere. The method does its job and produces a new `"JAVA"` object. But no variable points to this object; the result is lost. Because `lang` keeps pointing to the old object, the third line prints `java`. In the fourth line the result of the same call is assigned to `lang`; the arrow moves to the new object and the fifth line prints `JAVA`. Table 11.1 is the trace table of the five lines; Figure 11.4 shows the two critical moments.

Line	What does lang point to?	Objects on the heap	Output
1	"java"	"java"	
2	"java"	"java", "JAVA" (nobody points to it)	
3	"java"	"java", "JAVA" (garbage)	java
4	"JAVA" (new object)	"java", "JAVA"	
5	"JAVA"	"java", "JAVA"	JAVA

Table 11.1: The trace table of the toUpperCase example. The "Objects on the heap" column gives the string objects in memory after that line.

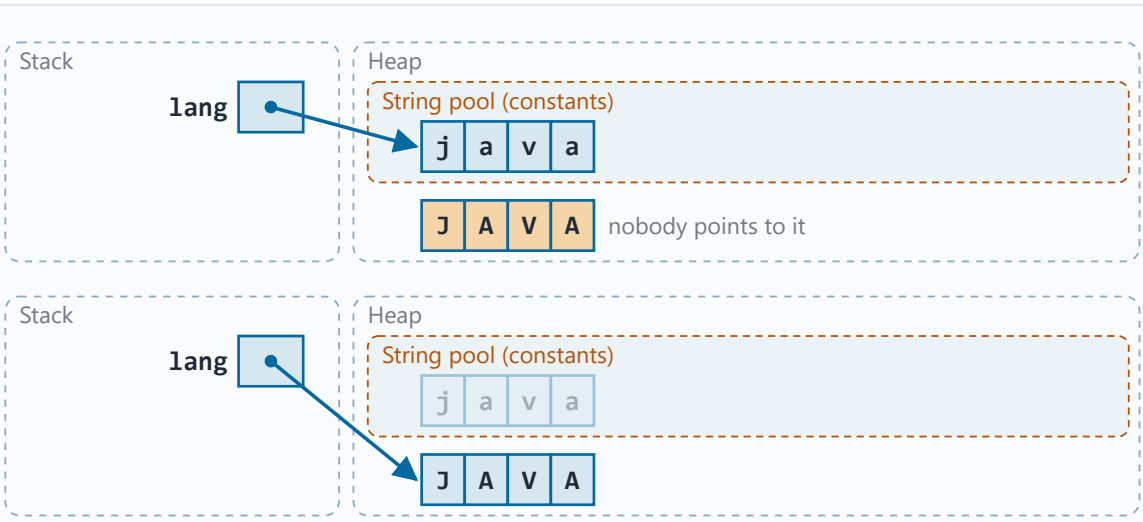


Figure 11.4: Top: the state after line 2, where `toUpperCase()` has produced a new "JAVA" object but no variable points to it. Bottom: the state after line 4, where the returned object has been assigned to `lang` and the arrow has moved to it. The "java" object is the same in both states.

Common mistake

Writing lines such as `name.trim();`, `name.toUpperCase();` or `name.replace('a', 'e');` on their own and thinking that `name` has changed. These lines compile and run, but the result is thrown away. The correct form is to assign the result again, as in `name = name.trim();`. Some development environments show a warning for calls whose returned value is not used.

There is also no way to change a single character of a string in place. For arrays you can write `a[0] = 5;`, but there is no corresponding operation for strings. `charAt` is for reading only; it cannot be assigned to:

```
public class SetCharAt {
    public static void main(String[] args) {
        String s = "java";
        s.charAt(0) = 'J';
        System.out.println(s);
    }
}
```

```
SetCharAt.java:4: error: unexpected type
        s.charAt(0) = 'J';
                ^
  required: variable
  found:    value
1 error
```

The message says that a variable was expected to the left of the equals sign but a value was found. `s.charAt(0)` produces a value; it is not a place in memory. To capitalize the first letter, a new string is built: `s = Character.toUpperCase(s.charAt(0)) + s.substring(1);` This line produces `"Java"`.

11.2.3 Why is immutability useful?

Producing a new object at every change may seem wasteful. Java's designers chose to pay this price for three gains:

- **Safe sharing.** Several variables can point to the same string object. Because none of them can change the object, an operation done through one variable does not affect the others. The string pool is possible because of this: all the `"Giresun"` literals in the program share a single object.
- **Predictability.** When a string is passed to a method, the method cannot corrupt it. The calling side is sure that the string is the same when it returns from the method.
- **Security.** File names, network addresses and passwords are carried in strings. That a file name that has passed a check cannot be changed afterward is an important guarantee.

The price is that a large number of concatenations slows down. If `s = s + x` is written thousands of times in a loop, a new object is produced on every iteration and the old contents are copied. For this case Java offers a mutable helper class: `StringBuilder`.

11.3 Indexing

The characters of a string are ordered, and each has a sequence number. This number is called the **index**. As with arrays, counting in strings starts at 0.

Definition 11.4

The characters of a string of length n are at **indices** 0, 1, ..., n − 1. The first character is at index 0 and the last at index n − 1. Reaching a character by its index is called **indexing**. In Java, `s.length()` returns the length of the string, and `s.charAt(i)` returns the character at index i as a `char` value.

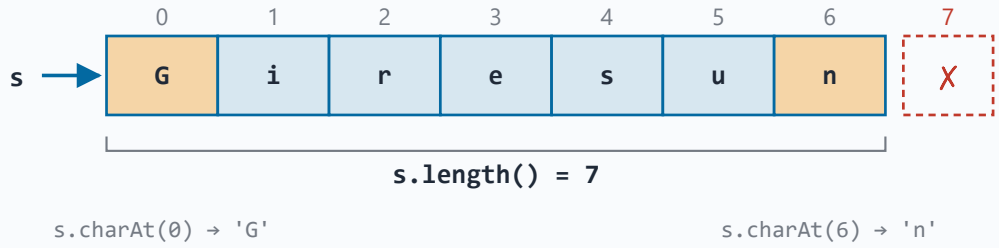


Figure 11.5: The indices of the string `s = "Giresun"`. The seven characters are numbered 0 to 6. `s.charAt(0)` returns 'G' and `s.charAt(6)` returns 'n'. There is no cell number 7; `s.charAt(7)` is invalid.

For the string in Figure 11.5, the following program prints the length and the first and last characters. The index of the last character is found for any string with the expression `s.length() - 1`; this way the code works whatever the length.

```
public class Index {
    public static void main(String[] args) {
        String s = "Giresun";
        System.out.println(s.length());
        System.out.println(s.charAt(0));
        System.out.println(s.charAt(s.length() - 1));
        int[] arr = {4, 8, 15};
        System.out.println(arr.length + " " + s.length());
        System.out.println(s.charAt(7));
    }
}
```

```
7
G
n
3 7
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: Index 7 out of
bounds for length 7
```

The last line tries to reach the nonexistent character number 7. The compiler does not catch this error, because the index is usually computed while the program runs. The error shows up at run time as a `StringIndexOutOfBoundsException`. The message says that index 7 is outside the bounds of a string of length 7. An index like this is called an **out-of-bounds** index.

The difference between `arr.length` and `s.length()` in the program is also important. For arrays, `length` is a field and is written without parentheses. For strings, `length()` is a method and is called with parentheses. If the parentheses are forgotten, the program does not compile:

```
public class StringNotMethod {
    public static void main(String[] args) {
        String s = "Giresun";
        System.out.println(s.length);
    }
}
```

```
StringNotMethod.java:4: error: cannot find symbol
        System.out.println(s.length);
                           ^
  symbol:   variable length
  location: variable s of type String
1 error
```

The compiler says that it cannot find a variable named `length` in the type `String`. The word “symbol” in the message means a named program element such as a variable, a method or a class.

Common mistake

Trying to reach the last character with `s.charAt(s.length())`. If the length is n, the last index is n − 1; `s.charAt(s.length())` is always out of bounds. The same mistake is seen in loops too: the last iteration of the loop `for (int i = 0; i <= s.length(); i++)` throws an exception at `charAt`. The correct condition is `i < s.length()`.

11.4 String methods

The `String` class has more than sixty methods. This section covers the ones most often needed in everyday programs. One rule holds for all of them: no method changes the original string. If a result is needed, the method returns it as a new value.

11.4.1 length, indexOf, substring and contains

Four basic methods are used together in this program:

```
public class BasicMethods {
    public static void main(String[] args) {
        String s = "Java Programlama";
        int len = s.length();
        int pos = s.indexOf("Programlama");
        String sub = s.substring(5, 16);
        boolean has = s.contains("Java");
        System.out.println(sub + " " + pos);
        System.out.println(len + " " + has + " " + s.indexOf("java") + " " + s.indexOf('a')
+ " " + s.lastIndexOf('a'));
    }
}
```

```
Programlama 5
16 true -1 1 15
```

The last `println` line of the program is a check line added to the first six lines; its output is explained separately below. The first six lines work as follows:

1. `s` points to the 16-character string `"Java Programlama"`. The space in between is also a character and is at index 4.
2. `s.length()` returns 16. The indices run from 0 to 15.
3. `s.indexOf("Programlama")` returns the index of the *first character* of the text searched for. "P" is at index 5; `pos` becomes 5. If the text searched for did not occur at all, the result would be −1.
4. `s.substring(5, 16)` gives the part that starts at index 5 and ends *before* index 16. The start is included and the end is excluded. The length of the part is *16 − 5 = 11*: `"Programlama"`.
5. `s.contains("Java")` returns `true` if the text searched for occurs anywhere in the string.
6. `sub + " " + pos` is concatenated and printed: `Programlama 5`.

Line	len	pos	sub	has	Output
1					
2	16				
3	16	5			
4	16	5	"Programlama"		
5	16	5	"Programlama"	true	
6	16	5	"Programlama"	true	Programlama 5

Table 11.2: The trace table of the basic methods program. An empty cell shows that the variable has not been declared yet.

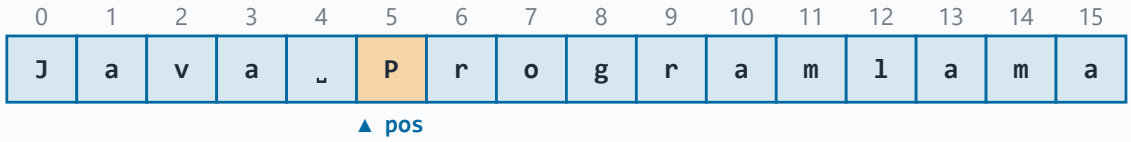


Figure 11.6: The sixteen cells of the string `"Java Programlama"`. `indexOf("Programlama")` gives the index of "P", the first character of the text searched for, that is, 5; the arrow points to this cell.

The check line added to the program shows three more details. `s.indexOf("java")` gives −1, because the search is case-sensitive: the string contains "Java", not "java". `s.indexOf('a')` gives 1; `indexOf` can also search for a single character and finds the place where it first occurs. `s.lastIndexOf('a')` searches from the end and finds the place where it last occurs, 15.

Example 11.2

For `String t = "bilgisayar";`, what are the values of `t.indexOf('a')`, `t.lastIndexOf('a')`, `t.indexOf("say")` and `t.substring(2, 5)`?

Solution. First write the indices: b=0, i=1, l=2, g=3, i=4, s=5, a=6, y=7, a=8, r=9. The first 'a' is at index 6 and the last 'a' at index 8. The text "say" starts with the 's' at index 5; the result is 5. `substring(2, 5)` takes the characters at indices 2, 3 and 4: `"lgi"`. Its length is *5 − 2 = 3*.

11.4.2 Frequently used methods

Table 11.3 summarizes the most frequently used methods. The program that follows tries all of them on short strings.

Method	Result	Return type
`length()`	Number of characters (code units)	`int`
`charAt(i)`	The character at position i	`char`
`substring(a, b)`	The substring from a (included) to b (excluded)	`String`
`indexOf(x)`	The first position of x; −1 if absent	`int`
`toLowerCase()`, `toUpperCase()`	A new string in lowercase or uppercase	`String`
`trim()`, `strip()`	A new string with leading and trailing whitespace removed	`String`
`isEmpty()`, `isBlank()`	Is the length 0; does it consist only of whitespace?	`boolean`

Table 11.3: Frequently used String methods. All return a result without changing the original string.

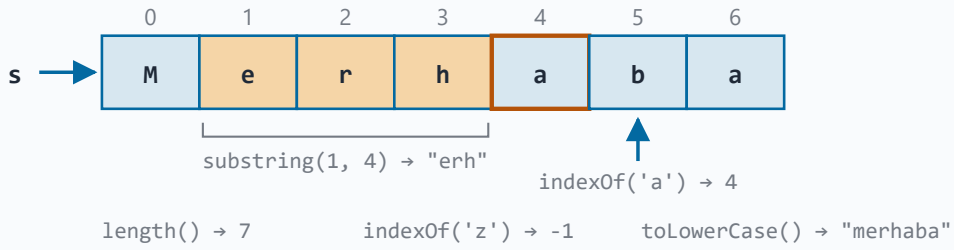


Figure 11.7: Five methods on `s = "Merhaba"`. `substring(1, 4)` takes indices 1, 2 and 3: "erh". `indexOf('a')` gives the index of the first 'a', 4; since 'z' never occurs, `indexOf('z')` is −1.

```
public class CommonMethods {
    public static void main(String[] args) {
        String s = "Merhaba";
        System.out.println(s.length());
        System.out.println(s.charAt(1));
        System.out.println(s.substring(1, 4));
        System.out.println(s.indexOf('a'));
        System.out.println(s.indexOf('z'));
        System.out.println(s.toLowerCase());
        System.out.println(s.toUpperCase());
        String k = "  Ada Can \t";
        System.out.println("[" + k.trim() + "] [" + k.strip() + "]");
        String g = " Ada ";
        System.out.println(g.trim().length() + " " + g.strip().length());
        System.out.println("".isEmpty() + " " + "   ".isEmpty() + " " + "   ".isBlank());
        System.out.println("ab".repeat(3));
    }
}
```

```
7
e
erh
4
-1
merhaba
MERHABA
[Ada Can] [Ada Can]
5 3
true false true
abababab
```

The first seven lines confirm the values in Figure 11.7. The following lines try the whitespace-removing methods. The string `k` has two spaces at the start and a space and a tab character (`\t`) at the end. `trim()` and `strip()` both remove these and do not touch the space in between: `[Ada`

`Can]`.

The difference between the two methods shows up with rarer characters. `trim()` removes only characters with codes of 32 and below, that is, ASCII spaces and control characters. `strip()` recognizes all of Unicode's whitespace characters. The string `g` has a wide space called the “em space” (U+2003) at both ends. `trim()` does not recognize this character and the length stays 5; `strip()` removes it and the length drops to 3. For user input and for copied and pasted text, `strip()` is safer.

`isEmpty()` gives `true` only for a string of length 0. `isBlank()` gives `true` for the empty string and for a string made up only of whitespace. So for the three-space string `" "`, the first is `false` and the second is `true`. The `repeat(3)` in the last line writes the string three times side by side: `ababab`.

11.4.3 Splitting with split

A list separated by commas, a sentence separated by spaces or a date with dots is processed by breaking it into parts. The character that separates the parts is called the **delimiter**. The `split` method cuts the string at the places where the delimiters are and returns the parts in a `String` array.

```
import java.util.Arrays;

public class SplitReplace {
    public static void main(String[] args) {
        String[] p = "a,b,,c".split(",");
        System.out.println(Arrays.toString(p) + " " + p.length); // [a, b, , c] 4
        String s = "ali veli";
        String t = s.replace('l', 'L');
        System.out.println(t); // aLi veLi
        System.out.println(s); // ali veli
        String[] q = "a,b,,".split(",");
        System.out.println(Arrays.toString(q) + " " + q.length);
        String[] r = ",a,b".split(",");
        System.out.println(Arrays.toString(r) + " " + r.length);
        String[] u = "3.14".split(".");
        System.out.println(u.length);
        String[] v = "3.14".split("\\.");
        System.out.println(Arrays.toString(v));
    }
}
```

```
[a, b, , c] 4
aLi veLi
ali veli
[a, b] 2
[, a, b] 3
0
[3, 14]
```

The first six lines of the program are the example from the slides; the following lines try the other rules of `split`. `"a,b,,c".split(",")` cuts the text at three commas and produces four parts: `"a"`, `"b"`, `""` and `"c"`. Between two consecutive commas there is nothing; this empty string also goes into the array. `Arrays.toString(p)` prints the array in readable form: `[a, b, , c]`. The gap between the two commas is where the empty part sits. Figure 11.8 shows this split.

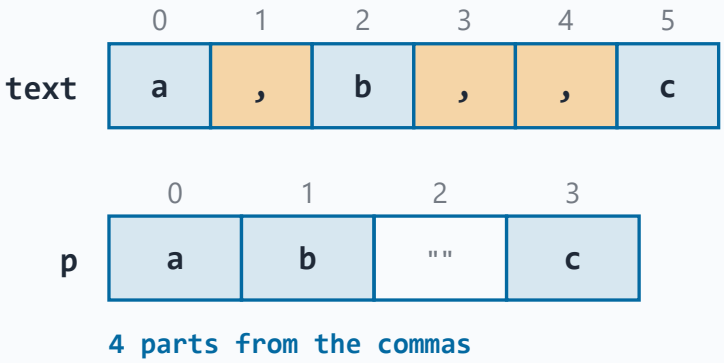


Figure 11.8: `"a,b,,c".split(",")`. The six-character text above is cut at the commas at indices 1, 3 and 4. The array `p` below has four cells; the cell at index 2 is the empty string.

The additional lines show three rules of `split`:

- **Trailing empty parts are dropped.** `"a,b,,".split(",")` gives two parts, not four: `[a, b]`. The two trailing empty parts do not go into the array.
- **A leading empty part is kept.** `",a,b".split(",")` gives three parts: `[, a, b]`. The empty text before the first comma counts as a part.
- **The delimiter is a regular expression.** The text given to `split` is not a plain character but a pattern. In the pattern language, the dot means “any character”. So `"3.14".split(".")` cuts at every character, all the parts are empty, and once the trailing empty parts are dropped the array length is 0. To split at the dot itself, `split("\\.")` is written and the result is `[3, 14]`.

Definition 11.5

A **regular expression** is a small language that describes a text pattern. Some characters have a special meaning in this language: `.` means any character, `+` means "one or more", `[0-9]` means any digit and `\s` means any whitespace character. To use a special character with its own meaning, a backslash is put in front of it. Because the backslash itself is written `\\` in a Java string literal, the pattern takes the form `"\\."` or `"\\s+"`.

Example 11.3

How many parts do `"23.10.2026".split("\\.")` and `"elma, armut,kiraz".split(",")` give, and what are the parts?

Solution. The first string has two dots; three parts come out: `"23"`, `"10"`, `"2026"`. The second string has two commas; again three parts come out: `"elma"`, `" armut"`, `"kiraz"`. The space at the start of the second part is kept, because the delimiter is only the comma. To remove the space too, either `strip()` is applied to each part or the delimiter is given as `",\\s*"` (a comma followed by zero or more spaces).

11.4.4 replace and regular expressions

`replace` returns a new string in which all the matches in a string have been replaced with another text. Lines 4–7 of the same program show this: `s.replace('l', 'L')` turns the two 'l' letters in `"ali veli"` (indices 1 and 6) into uppercase and returns `"aLi veLi"`. `s` stays the same; the last line again prints `ali veli`. Figure 11.9 shows the two objects side by side.

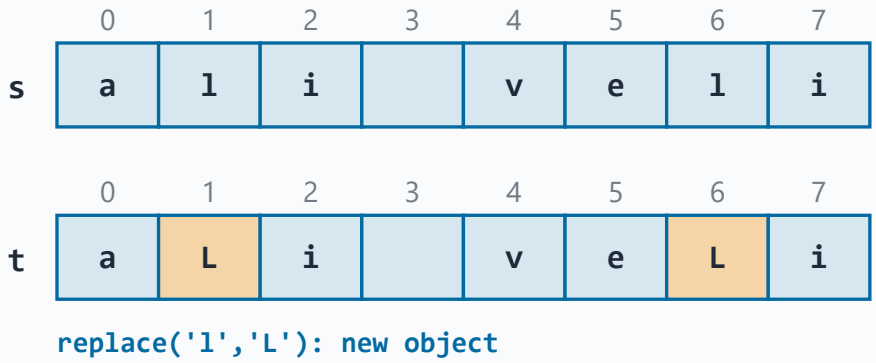


Figure 11.9: `t = s.replace('l', 'L')`. Top: the "ali veli" object that `s` points to stays unchanged. Bottom: `t` points to the new object with 'L' at indices 1 and 6.

The family of replacement methods has four members, and the difference between them lies in the kind of thing being searched for. Table 11.4 gives the examples from the slides together with JDK 21 output.

Call	Result	Explanation
`"banana".replace('a', 'o')`	`bonono`	every 'a' character becomes 'o'
`"banana".replace("an", "AN")`	`bANANa`	plain-text search; left to right, without overlap
`"a1b22c333".replaceAll("[0-9]+", "#")`	`a#b#c#`	each group of digits becomes a single #
`"a1b22c333".replaceFirst("[0-9]+", "#")`	`a#b22c333`	only the first match is replaced
`"Ada Can Ece".replaceAll(" +", " ")`	`Ada Can Ece`	consecutive spaces are reduced to one

Table 11.4: The replacement methods. The result column is the real output of the program.

`replace` searches for plain text, while `replaceAll` and `replaceFirst` search for a regular expression. The pattern `[0-9]+` means "one or more digits". In `"a1b22c333"` there are three groups of digits: `1`, `22` and `333`. Each group matches as a whole and is replaced with a single `#`. The pattern `" +"` means "one or more spaces"; gaps of two and three spaces are reduced to one space.

> **Common mistake**
>
> Using `replaceAll` to replace plain text. If the text searched for contains special characters such as the dot, plus or star, the result is unexpected. `"1.5".replace(".", ",")` gives the expected `1,5`. `"1.5".replaceAll(".", ",")`, however, replaces every character and produces `,,,`; the dot means "any character". For plain text `replace` is enough, and it too replaces all the matches.

11.4.5 join, valueOf and format

Three `static` methods are called with the class name, not through an object: `String.join`, `String.valueOf` and `String.format`.

- `String.join(", ", "Ada", "Can", "Ece")` joins the parts with a delimiter between them: `Ada, Can, Ece`. It can be thought of as the opposite of `split`.
- `String.valueOf(3.5)` converts a value to a string: `"3.5"`. If a character array is given, it joins the characters: `String.valueOf(new char[]{'h', 'i'})` returns `"hi"`.
- `String.format` uses the same format string as `printf` but does not print the result to the screen; it returns it as a string.

Table 11.5 gives three examples of `format` with two different locales. The whole program is the class `ReplaceTable`; it was run once on a Java virtual machine with an English (US) locale and once on one with a Turkish locale.

Call	on an en_US machine	on a tr_TR machine
`String.format(Locale.US, "%.2f", 3.14159)`	`3.14`	`3.14`
`String.format(Locale.US, "%5d|%-5d|%05d", 42, 42, 42)`	`␣␣42|42␣␣|00042`	`␣␣42|42␣␣|00042`
`String.format(Locale.US, "%,d", 1234567)`	`1,234,567`	`1,234,567`

Call	on an en_US machine	on a tr_TR machine
`String.format("%.2f", 3.14159)`	`3.14`	`3,14`
`String.format("%,d", 1234567)`	`1,234,567`	`1.234.567`

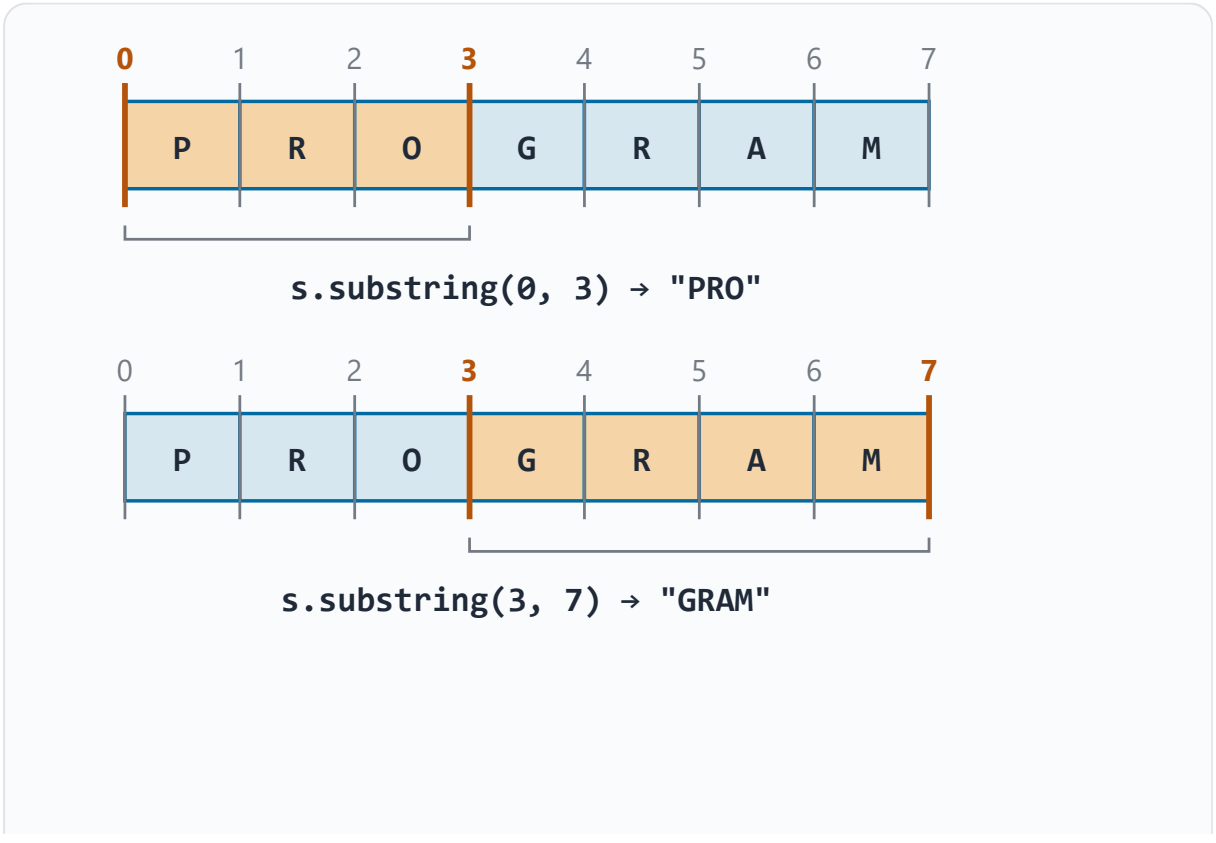
Table 11.5: String.format examples. In the last two rows no locale is given; the result depends on the machine's setting.

The ␣ mark in the table is there to make the spaces in the results visible. `%5d` right-aligns the number in a field of 5 characters; since 42 is two characters, three spaces come before it. `%-5d` left-aligns and puts the spaces at the end. `%05d` pads with zeros instead of spaces. `%,d` adds a thousands separator. `%.2f` rounds the number to two digits after the decimal point.

The rows given `Locale.US` give the same result on every machine. The last two rows, with no locale given, follow the language of the machine: on a computer with a Turkish setting the decimal separator is a comma and the thousands separator a dot. For numbers going to a file or to another program, the locale is given explicitly; for numbers shown to a user on the screen, the machine's setting may be appropriate.

11.4.6 Substring bounds

`substring` is the method on which mistakes are most often made. An easy way to remember the rule is to write the indices not on the characters but on the *lines between them*. In Figure 11.10, there are eight lines, from 0 to 7, among the seven characters of the string `"PROGRAM"`. `substring(a, b)` takes the characters that lie between line a and line b.



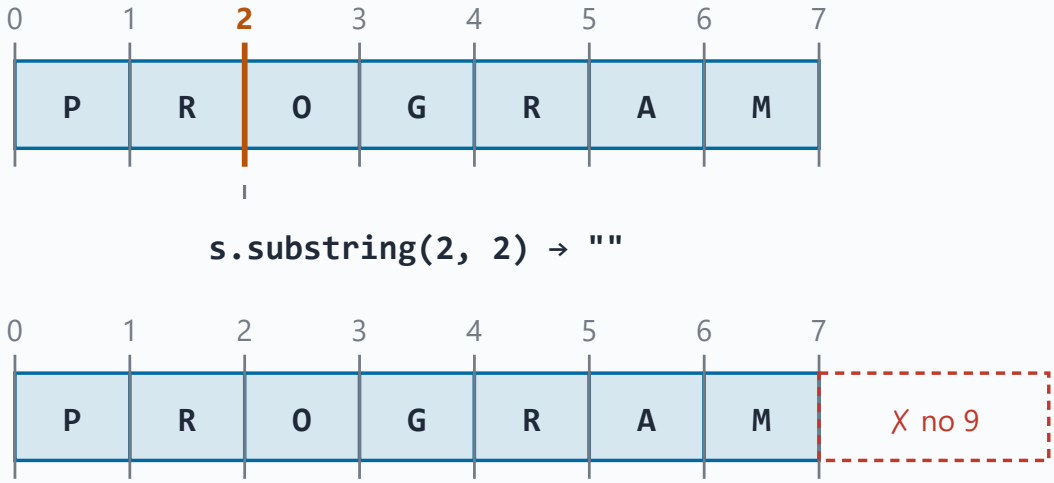


Figure 11.10: Four calls on `s = "PROGRAM"`. Between lines 0 and 3 is "PRO". The end can be `length()` = 7: between 3 and 7 is "GRAM". If the start equals the end, there are no characters in between: "". Because there is no line 9, `substring(5, 9)` gives an error.

Definition 11.6

A **substring** is a part of a string made up of consecutive characters. `s.substring(a, b)` returns the substring that starts at the character at index a and ends before the character at index b; its length is b − a. The valid range is $0 \le a \le b \le s.length()$. `s.substring(a)` gives the part from index a to the end of the string and is the same as `s.substring(a, s.length())`.

```
public class Substrings {
    public static void main(String[] args) {
        String s = "PROGRAM";
        System.out.println(s.substring(0, 3));
        System.out.println(s.substring(3, 7));
        System.out.println(s.substring(3));
        System.out.println("[" + s.substring(2, 2) + "]");
        System.out.println("[" + s.substring(7) + "]");
        System.out.println(s.substring(0, 3).length() + " " + s.substring(2, 5));
        System.out.println(s.substring(5, 9));
    }
}
```

```
PRO
GRAM
GRAM
[]
[]
3 OGR
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: Range [5, 9) out of
bounds for length 7
```

The output confirms each rule. `substring(0, 3)` takes three characters: *3 − 0 = 3*. `substring(3, 7)` and `substring(3)` give the same result, because the length of the string is 7. `substring(2, 2)` is the empty string; this is not an error, it means “take zero characters”. `substring(7)` is also the empty string: 7 is the last line and there are no characters after it. The `substring(5, 9)` on the last line throws an exception, because the end exceeds the length. The message says that the range [5, 9) is invalid for a string of length 7. The square bracket shows that the start is included and the round bracket that the end is excluded; this is the half-open interval notation of mathematics.

If the start is greater than the end, an exception is thrown too:

```
public class SubstringReversed {
    public static void main(String[] args) {
        String s = "PROGRAM";
        System.out.println(s.substring(5, 3));
    }
}
```

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: Range [5, 3) out of
bounds for length 7
```

Call	Condition 0 ≤ a ≤ b ≤ 7	Result	Length
`s.substring(0, 3)`	0 ≤ 0 ≤ 3 ≤ 7 holds	"PRO"	3
`s.substring(2, 5)`	holds	"OGR"	3
`s.substring(3, 7)`	holds	"GRAM"	4
`s.substring(2, 2)`	holds (a = b)	""	0
`s.substring(7)`	holds (a = 7)	""	0
`s.substring(5, 9)`	9 ≤ 7 is false	exception	–
`s.substring(5, 3)`	5 ≤ 3 is false	exception	–

Table 11.6: The substring calls for s = "PROGRAM" (length 7).

Common mistake

Taking the second parameter of `substring` to be "how many characters to take". `"PROGRAM".substring(2, 3)` gives not the three characters starting at index 2 but only one character ("O"). The second parameter is the end position, not the length. To take k characters, write `substring(a, a + k)`.

11.4.7 Application: taking an email address apart

When `indexOf` and `substring` are used together, a text can be split on the two sides of a delimiter. An email address is split by the `@` sign into a username and a domain name:

```
String email = "ali@gru.edu";
int at = email.indexOf('@');
String user = email.substring(0, at);
String domain = email.substring(at + 1);
char first = email.charAt(0);
System.out.println(user + " / " + domain);
System.out.println(Character.toUpperCase(first));
```

```
ali / gru.edu
A
```

In seven lines the program does this: it finds the position of `@`, takes what comes before and after it separately, reads the first character and converts it to uppercase. Table 11.7 is the trace table; Figure 11.11 shows the two cutting points.

Line	Operation	Variable	Output
2	`indexOf('@')`: @ is at index 3	at = 3	
3	`substring(0, 3)`: indices 0, 1, 2	user = "ali"	
4	`substring(4)`: from 4 to the end	domain = "gru.edu"	
5	`charAt(0)`	first = 'a'	
6	concatenation		ali / gru.edu
7	`Character.toUpperCase('a')`		A

Table 11.7: The trace table of the email program. email = "ali@gru.edu" (11 characters, indices 0–10).

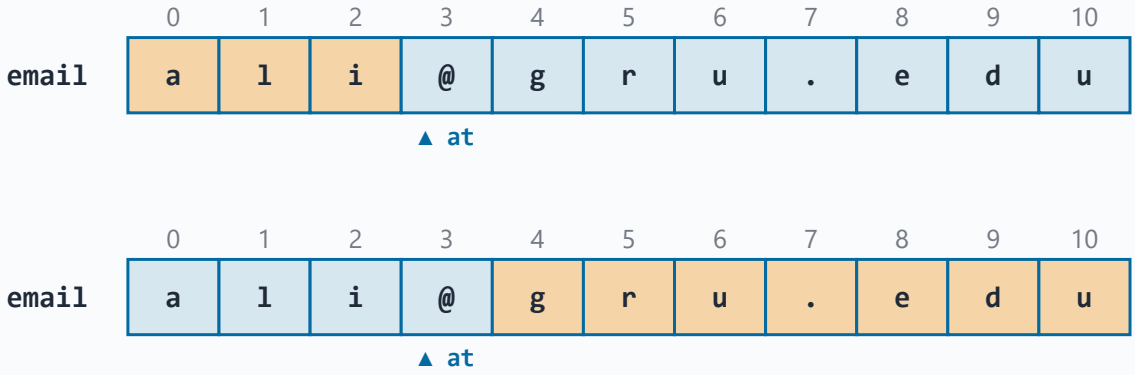


Figure 11.11: Splitting the string "ali@gru.edu". Top: `substring(0, at)`, indices 0–2 to the left of @, "ali". Bottom: `substring(at + 1)`, from index 4 after @ to the end, "gru.edu". The @ character itself is in neither part.

Two details deserve attention. The first is the reason for writing `at + 1`: if `substring(at)` were written, the domain would be `"@gru.edu"`. The second is the return type of `charAt`. `first` is a `char`, and to convert it to uppercase, the `toUpperCase` method of the `Character` class, not of `String`, is used. This method takes a `char` and returns a `char`.

The weak point of the program is the case where there is no `@` in the address. Then `indexOf` returns −1 and the call `substring(0, -1)` throws an exception:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: Range [0, -1) out of
bounds for length 10
```

A robust program first checks the result of the search:

```
public class EmailSafe {
    public static void main(String[] args) {
        String[] addresses = {"ali@gru.edu", "aligru.edu"};
        for (String email : addresses) {
            int at = email.indexOf('@');
            if (at < 0) {
                System.out.println(email + ": no @");
            } else {
                System.out.println(email.substring(0, at) + " / " + email.substring(at +
1));
            }
        }
    }
}
```

```
ali / gru.edu
aligru.edu: no @
```

Example 11.4

To produce the output “KAYA, ece” from the address `s = "ece.kaya@uni.tr"`, which indices are used?

Solution. The address has 15 characters; the indices are 0–14. `s.indexOf('@')` is 8; `s.substring(0, 8)` gives `"ece.kaya"`. In this new string the dot is at index 3. `substring(0, 3)` gives `"ece"` and `substring(4)` gives `"kaya"`. `"kaya".toUpperCase()` returns a new string `"KAYA"`. When the program is run, the output is:

```
KAYA, ece
```

11.5 Comparison

Asking whether two strings are the same is one of the most frequent jobs in programming: is the entered password correct, did the user type “yes”, do two students have the same name? For numbers, this question is asked with `==`. For strings, `==` does not ask the expected question.

11.5.1 == and equals

Because a `String` variable holds a reference, `a == b` compares two references. The question becomes “do the two variables point to the same object?”. Even if two separate objects carry the same characters, `==` gives `false`. To compare contents, the `equals` method is used.

Definition 11.7

Reference comparison (`a == b`) tests whether two variables point to the same object. **Content comparison** (`a.equals(b)`) tests whether two strings have the same length and carry the same characters at the same positions. The equality of strings is always asked with `equals`.

```
String a = "Giresun";
String b = new String("Giresun");
String c = "Giresun";
System.out.println(a == b);      // false
System.out.println(a == c);      // true
System.out.println(a.equals(b)); // true
```

```
false
true
true
```

The program creates three variables and makes three comparisons. Figure 11.12 gives the final state of memory, and Table 11.8 the line-by-line trace.

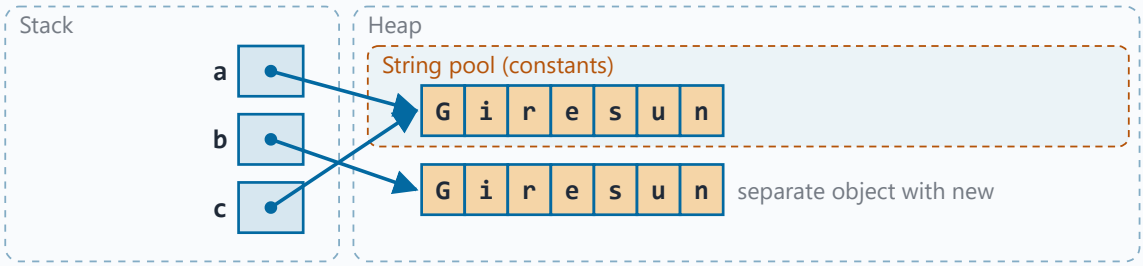


Figure 11.12: Memory after the three lines. `a` and `c` point to the single "Giresun" object in the string pool. `b` points to a separate "Giresun" object created on the heap with `new`. The two objects have the same contents and different locations.

Line	Operation	Result	Reason
1	`a = "Giresun"`	a → pool	placed in the constant pool
2	`b = new String("Giresun")`	b → new object	`new` always creates a separate object
3	`c = "Giresun"`	c → pool	the same constant; the object in the pool is shared
4	`a == b`	false	the two arrows go to different objects
5	`a == c`	true	both arrows are on the same object
6	`a.equals(b)`	true	the characters are equal one by one: G=G, i=i, …, n=n

Table 11.8: The trace table of the == and equals program.

The `true` on line 5 is misleading. That `a == c` comes out true is due not to the contents being equal but to the two constants sharing the same object in the pool. If one of the strings had been read from the keyboard, come from a file or been produced by an operation, it would be a separate object and `==` would give `false`. Where a string comes from is not always clear from looking at the code. So the rule is simple: for contents, `equals`.

Common mistake

Checking the answer typed by the user with `if (answer == "yes")`. The string read by `Scanner` is always a new object; even if the user types "yes", the condition comes out false. The program compiles and gives no error; it just behaves wrongly. The correct form is `if (answer.equals("yes"))`.

11.5.2 Comparison methods

Besides `equals`, four more comparison methods are often used. Table 11.9 gives them with examples; the results are the real output of the program.

Method	Purpose	Example	Result
`equals`	content equality	`"abc".equals("abc")`	true
`equalsIgnoreCase`	equality that ignores letter case	`"Java".equalsIgnoreCase("JAVA")`	true
`startsWith`	starts with a given text	`"Giresun".startsWith("Gir")`	true
`endsWith`	ends with a given text	`"Giresun".endsWith("sun")`	true
`compareTo`	lexicographic order: negative, 0 or positive	`"elma".compareTo("armut")`	4

Table 11.9: Comparison methods.

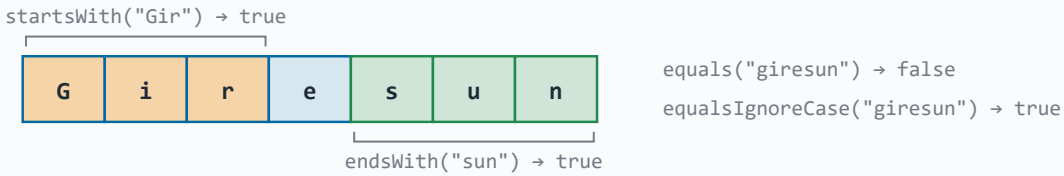


Figure 11.13: The string "Giresun" starts with "Gir" and ends with "sun". `equals("giresun")` gives `false`, because 'G' and 'g' are different characters. `equalsIgnoreCase("giresun")` gives `true`, because it ignores letter case.

```
public class ComparisonMethods {
    public static void main(String[] args) {
        System.out.println("abc".equals("abc"));
        System.out.println("Java".equalsIgnoreCase("JAVA"));
        System.out.println("Giresun".startsWith("Gir"));
        System.out.println("Giresun".endsWith("sun"));
        System.out.println("elma".compareTo("armut"));
        System.out.println("Giresun".equals("giresun"));
        System.out.println("Giresun".equalsIgnoreCase("giresun"));
        System.out.println("Giresun".startsWith("gir"));
    }
}
```

```
true
true
true
true
4
false
true
false
```

The last three lines show case sensitivity. `equals`, `startsWith` and `endsWith` compare characters as they are: `"Giresun".startsWith("gir")` is `false`. Where letter case does not matter (such as a command being typed as "YES", "Yes" or "yes"), `equalsIgnoreCase` is used.

11.5.3 compareTo and lexicographic order

Besides equality, the order of two strings can be asked too: which one comes first in a dictionary? This order is called **lexicographic order**. `a.compareTo(b)` returns an integer. If the result is negative, `a` comes first; if zero, the two are equal; if positive, `a` comes later.

Definition 11.8

`a.compareTo(b)` compares the two strings character by character, starting at the beginning. There are two cases:

1. If the first differing character is at position k, the result is *a.charAt(k) − b.charAt(k)*; that is, the difference of the codes of the two characters.
2. If all the characters of the shorter string are the same as the beginning of the longer one, the result is *a.length() − b.length()*; that is, the length difference.

If the two strings are exactly the same, the second rule gives 0.

```
String x = "elma";
String y = "armut";
System.out.println(x.compareTo(y)); // 4
String k = "kedi";
System.out.println(k.compareTo("kediler")); // -3
```

The two examples in the slides exercise the two rules separately. Figure 11.14 shows the critical step of each.

- `"elma".compareTo("armut")`: at position 0, 'e' meets 'a' and they differ. The result is *'e' − 'a' = 101 − 97 = 4*. Because it is positive, "elma" comes after "armut". The lengths of the strings (4 and 5) do not enter the result at all; the first differing character decides.

- `"kedi".compareTo("kediler")` : in the first four positions k=k, e=e, d=d, i=i. Then “kedi” ends; no differing character has been found. The result is the length difference: *4 − 7 = −3*. Because it is negative, the shorter “kedi” comes first.

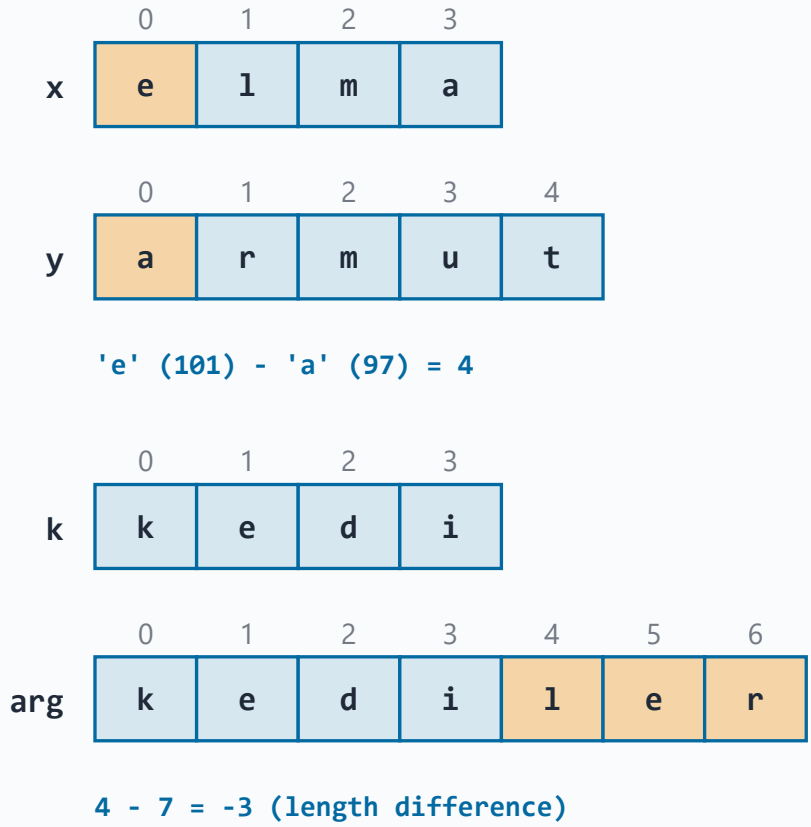


Figure 11.14: The two compareTo rules. Top: “elma” and “armut”: the first characters differ and the result is the code difference, *101 − 97 = 4*. Bottom: “kedi” and “kediler”: the first four characters are equal and the result is the length difference, *4 − 7 = −3*.

The program below makes eight more comparisons besides the two lines in the slides:

```
public class CompareTo {
    public static void main(String[] args) {
        String x = "elma";
        String y = "armut";
        System.out.println(x.compareTo(y)); // 4
        String k = "kedi";
        System.out.println(k.compareTo("kediler")); // -3
        System.out.println(y.compareTo(x));
        System.out.println("Ali".compareTo("ali"));
        System.out.println("abc".compareTo("abc"));
        System.out.println("Zeytin".compareTo("armut"));
        System.out.println("çilek".compareTo("dut"));
        System.out.println("ılgın".compareTo("incir"));
        System.out.println("Ali".compareToIgnoreCase("ali"));
        System.out.println("10".compareTo("9"));
    }
}
```

```
4
-3
-4
-32
0
-7
131
200
0
-8
```

Table 11.10 explains each result by the rule.

Call	First difference	Calculation	Result
`"elma".compareTo("armut")`	position 0: e / a	101 − 97	4
`"kedi".compareTo("kediler")`	none	4 − 7	−3
`"armut".compareTo("elma")`	position 0: a / e	97 − 101	−4
`"Ali".compareTo("ali")`	position 0: A / a	65 − 97	−32
`"abc".compareTo("abc")`	none	3 − 3	0
`"Zeytin".compareTo("armut")`	position 0: Z / a	90 − 97	−7
`"çilek".compareTo("dut")`	position 0: ç / d	231 − 100	131
`"ılgın".compareTo("incir")`	position 0: ı / i	305 − 105	200
`"Ali".compareToIgnoreCase("ali")`	case is ignored	–	0
`"10".compareTo("9")`	position 0: 1 / 9	49 − 57	−8

Table 11.10: How the compareTo results are computed. The code column gives the Unicode codes of the first differing characters.

The lower half of the table shows where `compareTo` departs from true dictionary order:

- **Uppercase letters come before lowercase ones.** The codes of uppercase letters are in the range 65–90, and those of lowercase letters in 97–122. This is why "Zeytin" comes before "armut".
- **Turkish letters fall to the end.** The codes of letters such as 'ç' (231), 'ı' (305) and 'ş' (351) are greater than those of all English letters. "çilek" comes after "dut", and "ılgın" after "incir". In the Turkish alphabet, however, ç comes before d and ı before i.
- **Numbers are compared like text.** "10" comes before "9", because the first characters are '1' and '9'. To compare the values of numbers, they must first be converted to numbers with `Integer.parseInt`.

For sorting that fits the Turkish alphabet, Java's `java.text.Collator` class is used. A `Collator` knows the sorting rules of a language. The program below sorts the same list of fruits first in the order of `compareTo`, then with a Turkish `Collator`:

```
import java.text.Collator;
import java.util.Arrays;
import java.util.Locale;

public class Sorting {
    public static void main(String[] args) {
        String[] fruits = {"zeytin", "çilek", "armut", "Elma", "ıhlamur", "incir", "üzüm",
"dut"};
        String[] a = fruits.clone();
        Arrays.sort(a);
        System.out.println(Arrays.toString(a));
        String[] b = fruits.clone();
        Arrays.sort(b, Collator.getInstance(Locale.forLanguageTag("tr")));
        System.out.println(Arrays.toString(b));
    }
}
```

```
[Elma, armut, dut, incir, zeytin, çilek, üzüm, ıhlamur]
[armut, çilek, dut, Elma, ıhlamur, incir, üzüm, zeytin]
```

In the first line, "Elma", which starts with an uppercase letter, has moved to the front, and "çilek", "üzüm" and "ıhlamur", which start with Turkish letters, have fallen to the end. The second line is the order of the Turkish alphabet: a, ç, d, e, ı, i, ü, z. A `Collator` also treats the uppercase–lowercase difference as secondary; "Elma" takes the place of the letter e.

Example 11.5

What are the values of `"kalem".compareTo("kale")`, `"kale".compareTo("kalp")` and `"Kale".compareTo("kale")`?

Solution. In the first call “kale” is the beginning of “kalem”; there is no differing character. The result is the length difference: *5 − 4 = 1*. In the second call the first three characters are equal; at position 3, 'e' meets 'p'. The result is *101 − 112 = −11*. In the third call the first characters are 'K' (75) and 'k' (107). The result is *75 − 107 = −32*. The difference between the uppercase and lowercase forms of the same letter is 32 for all English letters.

11.5.4 Turkish characters and the locale

Converting between uppercase and lowercase is simple in most languages: a ↔ A, b ↔ B. In Turkish, however, there are two separate letters i, dotted and dotless: i ↔ İ and ı ↔ I. In English these pairs are i ↔ I. The lowercase of the same letter “I” is “i” in English and “ı” in Turkish. Java solves this difference with the **locale**. A locale is a `java.util.Locale` object that carries the writing rules of a language and region.

Definition 11.9

`toLowerCase(Locale)` and `toUpperCase(Locale)` do the letter conversion according to the rules of the given language. The locale for Turkish is obtained with `Locale.forLanguageTag("tr")` and for English with `Locale.ENGLISH`. The parameterless `toLowerCase()` and `toUpperCase()` use the default locale of the machine on which the program runs.

```
import java.util.Locale;

public class TurkishLocale {
    public static void main(String[] args) {
        Locale tr = Locale.forLanguageTag("tr");
        System.out.println("TITLE".toLowerCase(Locale.ENGLISH));
        System.out.println("TITLE".toLowerCase(Locale.forLanguageTag("tr")));
        System.out.println("i".toUpperCase(tr));
        System.out.println("i".toUpperCase(Locale.ENGLISH));
        System.out.println("I".toLowerCase(tr));
        System.out.println("ISPARTA".toLowerCase(tr));
        System.out.println("ısparta".toUpperCase(tr));
        System.out.println("TITLE".toLowerCase(tr).equals("title"));
        System.out.println("title".equalsIgnoreCase("TITLE"));
        System.out.println("istanbul".toUpperCase(tr) + " " +
"istanbul".toUpperCase(Locale.ENGLISH));
        System.out.println("İZMİR".toLowerCase(tr) + " " +
"İZMİR".toLowerCase(tr).length());
        System.out.println("İZMİR".toLowerCase(Locale.ENGLISH).length() + " " +
"İZMİR".length());
        System.out.println("ISPARTA".equalsIgnoreCase("ısparta") + " " +
"i".equalsIgnoreCase("İ"));
    }
}
```

```
title
tıtle
İ
I
ı
ısparta
ISPARTA
false
true
İSTANBUL ISTANBUL
izmir 5
7 5
true true
```

Table 11.11 explains the results of the first seven calls with the rules of the two languages.

Call	Result	Rule
`"TITLE".toLowerCase(Locale.ENGLISH)`	title	English: I → i
`"TITLE".toLowerCase(Locale.forLanguageTag("tr"))`	tıtle	Turkish: I → ı
`"i".toUpperCase(tr)`	İ	Turkish: i → İ
`"i".toUpperCase(Locale.ENGLISH)`	I	English: i → I

Call	Result	Rule
`"I".toLowerCase(tr)`	ı	Turkish: I → ı
`"ISPARTA".toLowerCase(tr)`	ısparta	Turkish: I → ı
`"ısparta".toUpperCase(tr)`	ISPARTA	Turkish: ı → I

Table 11.11: Turkish and English letter conversion. tr = Locale.forLanguageTag("tr").

The following lines show the effect of the conversion on equality tests. Because `"TITLE".toLowerCase(tr)` gives "tıtle", it is not equal to `"title"`. `equalsIgnoreCase`, on the other hand, compares characters one by one with a language-independent rule and gives `true`. "istanbul" becomes "İSTANBUL" under Turkish rules and "ISTANBUL" under English rules. The Turkish lowercase of "İZMİR" is the five-letter "izmir". When it is lowercased with English rules, however, each "İ" turns into an "i" plus a separate dot mark added on top of it. The result looks like "izmir" on the screen but has length 7. This text does not pass an `equals` test against "izmir".

The most insidious problem is in the parameterless calls. The program below runs the same code on two different machines: one has an English (US) locale and the other a Turkish one.

```
import java.util.Locale;

public class DefaultLocale {
    public static void main(String[] args) {
        System.out.println(Locale.getDefault());
        String file = "ARSIV.ZIP".toLowerCase();
        System.out.println(file);
        System.out.println(file.endsWith(".zip"));
        System.out.println("title".toUpperCase());
        System.out.println("ARSIV.ZIP".toLowerCase(Locale.ENGLISH).endsWith(".zip"));
        System.out.printf("%.2f%n", 17.5);
    }
}
```

On the machine with the English setting, the output is:

```
en_US
arsiv.zip
true
TITLE
true
17.50
```

On the machine with the Turkish setting, the same program prints:

```
tr_TR
arsıv.zıp
false
TİTLE
true
17,50
```

The program checks whether a file name ends with ".zip". On the English machine, "ARSIV.ZIP" becomes "arsiv.zip" when lowercased and the check passes. On the Turkish machine the same line produces "arsıv.zıp" and the check fails. There is no change in the code; only the language of the computer is different. `"title".toUpperCase()` also gives "TİTLE" on the Turkish machine. The fifth line, which gives the locale explicitly, prints `true` on both machines. The `printf` on the last line is also affected by the locale: the decimal separator is a dot on one machine and a comma on the other.

Common mistake

Converting technical text such as keywords, command names, file extensions and email addresses with the parameterless `toLowerCase()` or `toUpperCase()`. The program works correctly on the developer's computer and breaks on a computer with a Turkish setting. This bug is known in the software world as "the Turkish i problem". For technical text, `Locale.ENGLISH` (or the language-independent `Locale.ROOT`) is given explicitly, and for Turkish text `Locale.forLanguageTag("tr")`.

In older sources the Turkish locale is also created as `new Locale("tr")`. This constructor has been deprecated since JDK 19; the JDK 21 compiler gives a warning for this line with the `-Xlint` option. `Locale.forLanguageTag("tr")` or `Locale.of("tr")` is used instead.

Example 11.6

A program converts the name of a province typed by the user to uppercase and compares it with `"IĞDIR"`. The user types "ığdır". What is the result with `toUpperCase(Locale.ENGLISH)` and with `toUpperCase(tr)`?

Solution. The uppercase of the dotless "ı" is "I" in both languages; English rules also recognize this Turkish letter. Both calls give "IĞDIR" and the comparison passes. If the user had typed "iğdir", the situation would change: with Turkish rules it would give "İĞDİR" and with English rules "IĞDIR". In a Turkish text the Turkish rules are the correct ones; in this case it turns out that "iğdir" was misspelled.

11.6 Memory: the string pool

The same string literal can occur in many places in a program: “Giresun”, “yes”, the empty string. Creating a separate object for each would waste memory. This is why Java gathers the literals in a common pool.

11.6.1 The pool and new

Definition 11.10

The **string pool** is the region of memory in which the Java virtual machine keeps string literals. All literals with the same contents share a single object in the pool. `new String(...)`, however, does not look at the pool and creates a new object on the heap on every call.

Sharing is safe only because strings are immutable. If they could be changed, a change made through one variable would affect all the variables using the same literal. Figure 11.15 shows the state of three variables.

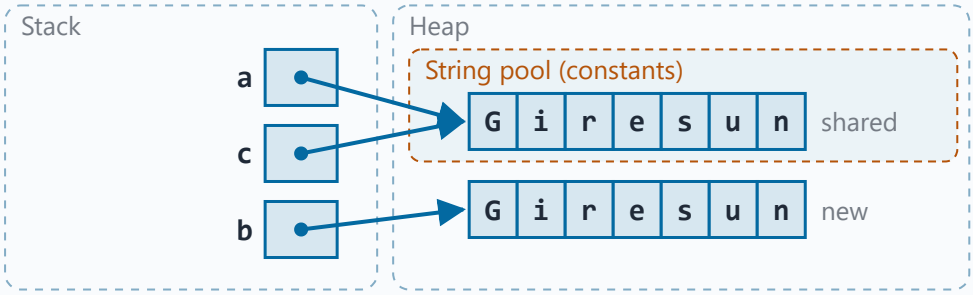


Figure 11.15: The string pool. Because `a` and `c` were created with the same literal, they share the single “Giresun” object in the pool. Because `b` was created with `new`, it points to a separate object on the heap.

```
public class Pool {
    public static void main(String[] args) {
        String a = "Giresun";
        String c = "Giresun";
        String b = new String("Giresun");
        String d = new String("Giresun");
        System.out.println((a == c) + " " + (a == b) + " " + (b == d));
        System.out.println(a.equals(b) + " " + b.equals(d));
        System.out.println((b.intern() == a) + " " + (b.intern() == b));
    }
}
```

```
true false false
true true
true false
```

The first line makes three reference comparisons. `a == c` is true: both point to the object in the pool. `a == b` is false: `b` is a separate object on the heap. `b == d` is false too: two `new` calls produce two separate objects. The `equals` tests on the second line are all true, because the contents are the same.

11.6.2 Constant concatenation and intern

Whether a string comes from the pool is not always easy to read from the code. The program below shows two subtle cases of this:

```
String a = "a", b = "b";
String r = a + b;
final String f = "a";
System.out.println(("a" + "b") == "ab");  // true
System.out.println((f + "b") == "ab");    // true
System.out.println(r == "ab");            // false
System.out.println(r.equals("ab"));       // true
System.out.println(r.intern() == "ab");   // true
```

```
true
true
false
true
true
```

The five results are explained as follows:

1. `("a" + "b") == "ab"` is true. Because both operands are constants, the compiler does the concatenation before the program runs. In the compiled code, the constant `"ab"` appears directly instead of `"a" + "b"`, and this constant comes from the pool.
2. `(f + "b") == "ab"` is true too. `f` is declared `final` and initialized with a constant. Because its value can never change, the compiler treats it as a constant as well.
3. `r == "ab"` is false. In the line `r = a + b`, `a` and `b` are ordinary variables. The concatenation is done at run time and the result is a new object on the heap.
4. `r.equals("ab")` is true; the contents are the same.
5. `r.intern() == "ab"` is true. The `intern()` method looks at the pool: if a string with the same contents exists there, it returns that one; if not, it adds the string to the pool and returns it. The returned reference is the `"ab"` in the pool.

Expressions whose value is known at compile time are called **constant expressions**: string literals, `final` constants and combinations of them. The result of a constant expression is put in the pool. Every concatenation that involves a variable produces a new object.

The effect of `intern` can also be seen in the last line of the `Pool` program. `b.intern() == a` is true: the contents of `b` exist in the pool, and `intern` returns the object in the pool. `b.intern() == b` is false: `b` itself is an object outside the pool. In practice `intern` is rarely needed; it is used here to make the workings of the pool visible.

> **Common mistake**
>
> Seeing `==` give the right result in a few trials and continuing to compare strings with `==`. Because the literals used in the trials are shared in the pool, `==` comes out right by coincidence. The same code gives a wrong result when the text comes from the keyboard or from a file. `==` being true shows not that the contents are equal but that the object is the same.

11.6.3 Null-safe comparison

`equals` is a method and is called through an object. If the variable is `null`, the call cannot be made and a `NullPointerException` occurs. Writing the same comparison the other way round removes this problem:

```
public class NullSafe {
    public static void main(String[] args) {
        String x = null;
        System.out.println("constant".equals(x));
        System.out.println(x.equals("constant"));
    }
}
```

```
false
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"String.equals(Object)" because "x" is null
```

In the line `"constant".equals(x)`, the method is called on `"constant"`, a real object. When the parameter is `null`, `equals` does not raise an error; it simply returns `false`. In the line `x.equals("constant")`, however, the method is called through `null` and the program stops. This is why the constant is written on the left: `"yes".equals(answer)` is safe even if `answer` is `null`.

11.7 Text input with Scanner

The `java.util.Scanner` class is used to read text from the keyboard. The class has two methods for reading text: `next()` reads a word and `nextLine()` reads a line. The key to understanding the behavior of the two methods is the **buffer**, where the characters coming from the keyboard wait.

11.7.1 next and nextLine

When the user types something and presses Enter, the characters typed and the end-of-line character go into the buffer. In these notes the end-of-line character is shown as ⏎. Each read method of `Scanner` takes characters from the start of the buffer and moves the reading position forward. Characters that have been read are not read again.

Definition 11.11

`next()` skips leading spaces and line ends, then reads the non-whitespace characters up to the first space or line end. This part is called a **token**. The space or ⏎ after the token stays in the buffer. `nextLine()`, on the other hand, reads everything from its current position to the end of the line, spaces included. It consumes the ⏎ but does not add it to the result.

```
import java.util.Scanner;
// ...
Scanner in = new Scanner(System.in);
String word = in.next();
in.nextLine();
String line = in.nextLine();
```

Figure 11.16 summarizes how this code works when the buffer holds "Ada⏎Merhaba dünya⏎". `next()` reads "Ada" and leaves the ⏎. The `nextLine()` in between consumes only that ⏎ and returns the empty string. The next `nextLine()` reads the whole second line.

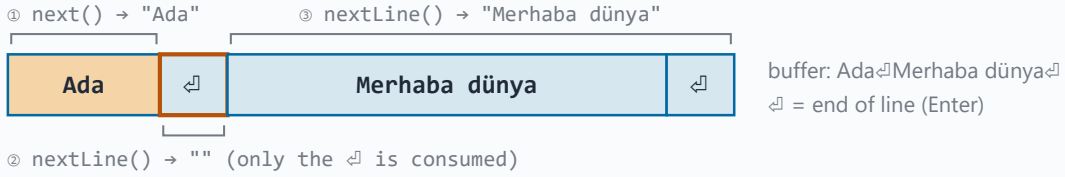


Figure 11.16: Three reads while the buffer is "Ada⏎Merhaba dünya⏎". ① `next()` reads "Ada". ② `nextLine()` consumes only the ⏎ and returns "". ③ `nextLine()` reads "Merhaba dünya".

That `next()` stops at whitespace makes it easy to read the words on a line one by one. "␣Ada␣␣Can␣Ece␣Su" is entered on a single line to the following program (␣ shows a space):

```
import java.util.Scanner;

public class NextWords {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String a = in.next();
        String b = in.next();
        String c = in.nextLine();
        System.out.println("[" + a + "] [" + b + "] [" + c + "]");
    }
}
```

```
[Ada] [Can] [ Ece Su]
```

The first `next()` skips the two leading spaces and reads "Ada". The second `next()` skips the three spaces in between and reads "Can". `nextLine()` takes the rest of the line as it is: the space at its start also goes into the result and "␣Ece␣Su" is read. `nextLine()` does not skip spaces.

11.7.2 Tracing the buffer step by step

The trace example in the slides is five lines long:

```
Scanner in = new Scanner(System.in);
String word = in.next();
in.nextLine();
String line = in.nextLine();
System.out.println(word + " | " + line);
```

The program is run once. The user first types "Ada" and presses Enter, then types "Giresun Merkez" and presses Enter. The output is:

```
Ada | Giresun Merkez
```

Step	Line	Event	Unread buffer	Result
1	1	`Scanner` is attached to the keyboard	(empty)	
2	2	`next()` finds nothing to read and waits; the user enters "Ada⏎"	Ada⏎	
3	2	`next()` reads up to a space or ⏎	⏎	word = "Ada"

Step	Line	Event	Unread buffer	Result
4	3	`nextLine()` reads up to ⏎ and consumes the ⏎	(empty)	"" (discarded)
5	4	`nextLine()` waits; the user enters "Giresun Merkez⏎"	Giresun␣Merkez ⏎	
6	4	`nextLine()` reads the whole line and consumes the ⏎	(empty)	line = "Giresun Merkez"
7	5	printing	(empty)	Ada | Giresun Merkez

Table 11.12: Tracing the Scanner buffer. The buffer column shows the characters after the reading position that have not been read yet.

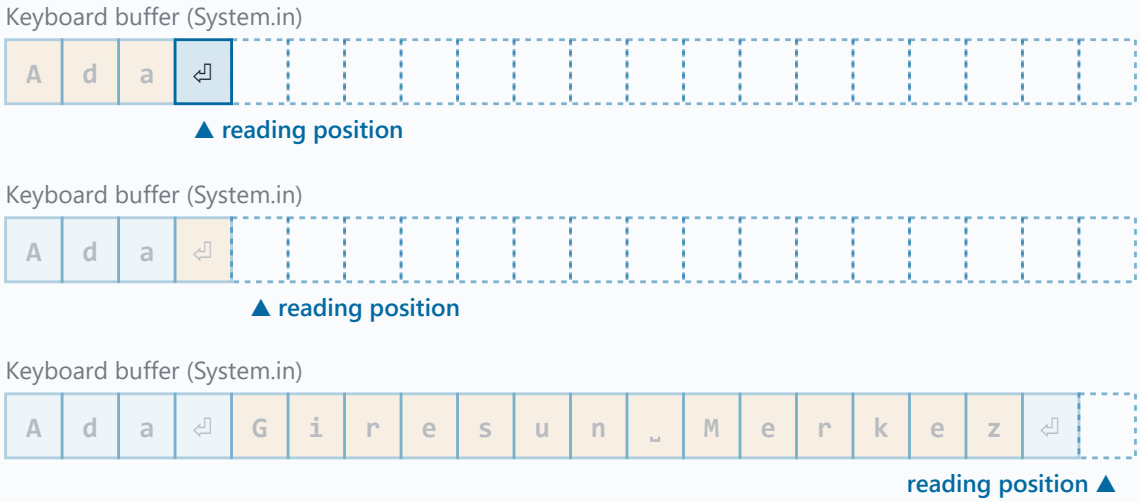


Figure 11.17: The keyboard buffer at steps 3, 4 and 6 of the table. Characters that have been read are faded, and the reading position is shown with ▲. At step 3, `next()` has read "Ada" and the ⏎ is left in the buffer. At step 4, `nextLine()` has consumed the ⏎. At step 6, the second line has also been read completely.

The result of the call `in.nextLine();` on line 3 is not assigned to any variable. The only job of this line is to clear the ⏎ left in the buffer. What happens if the line is deleted? The `nextLine()` on line 4 finds the leftover ⏎, returns the empty string at once and gives the user no chance to type the second line. The program below shows this; the input is again the same:

```
import java.util.Scanner;

public class BufferBug {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String word = in.next();
        String line = in.nextLine();
        System.out.println(word + "|" + line + "|");
    }
}
```

```
Ada||
```

`line` is the empty string; there is nothing between the two vertical bars. The "Giresun Merkez" the user typed has never been read and has stayed in the buffer.

11.7.3 nextLine after nextInt

The same problem appears after methods that read numbers, such as `nextInt()` and `nextDouble()`. These methods behave like `next()`: they read the number and leave the line end in the buffer. The program below wants to read the age first and then the full name. The user enters "19⏎" and "Ada Yılmaz⏎":

```
import java.util.Scanner;

public class AgeName {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Age: ");
        int age = in.nextInt();
        System.out.print("Full name: ");
        String name = in.nextLine();
        System.out.println();
        System.out.println("[" + name + "] " + age);
    }
}
```

```
Age: Full name:
[] 19
```

The "Full name:" question appears on the screen but the program does not wait for the answer. `nextLine()` reads the ⏎ left over after the 19, and `name` stays empty. The fix is to add a `nextLine()` that clears the leftover after the number:

```
import java.util.Scanner;

public class AgeNameFixed {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Age: ");
        int age = in.nextInt();
        in.nextLine();
        System.out.print("Full name: ");
        String name = in.nextLine();
        System.out.println();
        System.out.println("[" + name + "] " + age);
    }
}
```

```
Age: Full name:
[Ada Yılmaz] 19
```

You may notice that the line looks empty after the text “Full name:” in the output. The reason is that the input was supplied from a file: when run from the keyboard, the text typed by the user appears on that line. This is why `System.out.println()` was added to the program.

Common mistake

Using `nextInt()` and `nextLine()` one after the other and not understanding why the second read is “skipped”. The read is not skipped: `nextLine()` has read the empty line left in the buffer. The rule is this: if a line is to be read after `next()`, `nextInt()` or `nextDouble()`, a `nextLine()` that consumes the ⏎ is put in between. Another way is to read all the input line by line with `nextLine()` and convert the numbers with `Integer.parseInt`.

11.8 Concatenation and StringBuilder

The `+` operator is readable and sufficient for joining a few strings. But if concatenation is done thousands of times inside a loop, the price of immutability becomes noticeable.

11.8.1 The cost of using + in a loop

The line `s = s + x` does not lengthen `s`. Each time it runs, it produces a new object that carries all the characters of `s` and `x`. The old object goes to the garbage. Figure 11.18 shows this for a loop of three iterations.

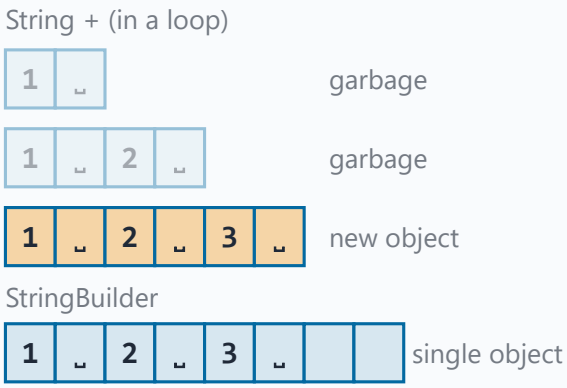


Figure 11.18: Top: concatenation with `+`; each iteration produces a new object named "1␣", "1␣2␣", "1␣2␣3␣"; the earlier ones become garbage. Bottom: `StringBuilder`, where a single object grows.

The number of characters copied can be computed. Consider a loop that appends a single character on each iteration. On iteration 1 a new object of 1 character is built, on iteration 2 one of 2 characters, and on iteration n one of n characters. The total number of characters copied is:

$$1 + 2 + 3 + \ldots + n = n(n + 1) / 2$$

This total is roughly $n^2/2$. For n = 1,000 it is 500,500, and for n = 100,000 it comes to about 5 billion characters. The program below verifies these numbers by counting inside the loop:

```
public class Cost {
    public static void main(String[] args) {
        for (int n : new int[]{1000, 10000, 100000}) {
            long copied = 0;
            String s = "";
            for (int i = 0; i < n; i++) {
                copied += s.length() + 1;
                s = s + "x";
            }
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < n; i++) {
                sb.append('x');
            }
            System.out.println(n + " " + copied + " " + (s.length() == sb.length()));
        }
    }
}
```

```
1000 500500 true
10000 50005000 true
100000 5000050000 true
```

When n grows tenfold, the number of characters copied grows about a hundredfold: from 500,500 to 50,005,000 and then to 5,000,050,000. This rate of growth is called **quadratic** growth, and for large values of n the program slows down noticeably. In the second loop, with `StringBuilder`, each character is written to the buffer once; the total work is proportional to n. The `true` at the end of the line shows that the two methods produced strings of the same length.

11.8.2 Concatenating with StringBuilder

> **Definition 11.12**
>
> `StringBuilder` is a class that keeps characters in a **mutable** buffer. The `append` method adds to the end of the buffer and does not produce a new object. When the contents of the buffer are finished, they are converted to a `String` with `toString()`. `StringBuilder` is also in the `java.lang` package.

```
StringBuilder text = new StringBuilder();
for (int i = 1; i <= 3; i++) {
    text.append(i).append(' ');
}
String result = text.toString();
System.out.println(result + "|");
```

```
1 2 3 |
```

Table 11.13 gives the ten steps of the program. The loop variable `i` runs from 1 to 3, and two appends are made on each iteration: the number and then a space.

Step	Line	i	Condition / action	text (buffer)	result
1	1		an empty buffer is created	""	
2	2	1	1 <= 3 → true	""	
3	3	1	`append(1).append(' ')`	"1␣"	
4	2	2	i++; 2 <= 3 → true	"1␣"	
5	3	2	`append(2).append(' ')`	"1␣2␣"	
6	2	3	i++; 3 <= 3 → true	"1␣2␣"	
7	3	3	`append(3).append(' ')`	"1␣2␣3␣"	
8	2	4	i++; 4 <= 3 → false, the loop ends	"1␣2␣3␣"	
9	5		`toString()`	"1␣2␣3␣"	"1␣2␣3␣"

Step	Line	i	Condition / action	text (buffer)	result
10	6		printing: `result + "|"`	"1␣2␣3␣"	"1␣2␣3␣"

Table 11.13: The trace table of the StringBuilder program. ␣ is the space character.

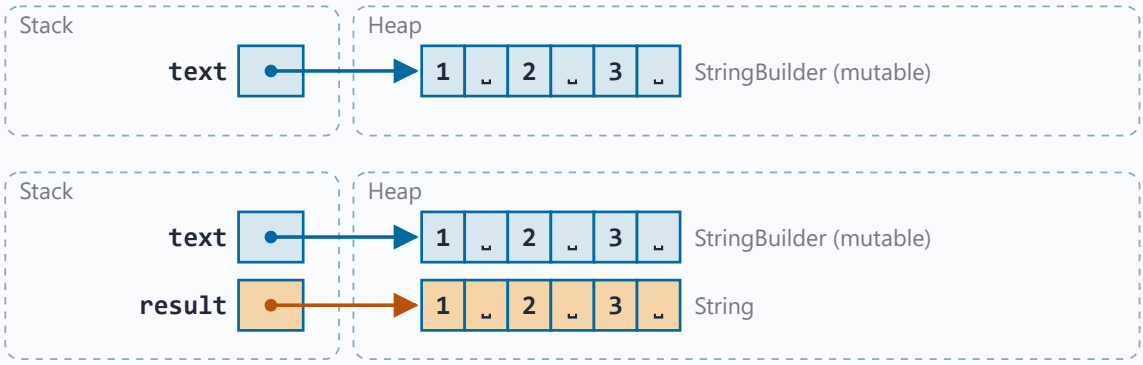


Figure 11.19: Top: after step 7, a single `StringBuilder` object carries the characters "1␣2␣3␣"; no new object has been created during the loop. Bottom: after step 9, `toString()` has produced a `String` object separate from the current contents of the buffer, and `result` points to it.

The form `text.append(i).append(' ')` on line 3 is noteworthy. `append` returns the very object it appended to. That is why the second `append` can be called on the same object. Calls in a row like this are called **chaining**. The `"|"` at the end is there only to make the space at the end of the output visible.

`append` accepts a value of any type: `int`, `char`, `double`, `boolean`, `String`. Numbers are appended in their text form. `append(1)` adds the character 1 to the buffer; it does not perform an addition.

11.8.3 The other methods of StringBuilder

Because `StringBuilder` is mutable, it also has methods that `String` does not. These change the buffer in place:

```
public class SbMethods {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Java");
        sb.append(21);
        System.out.println(sb + " " + sb.length());
        sb.insert(0, "JDK ");
        System.out.println(sb);
        sb.setCharAt(4, 'j');
        System.out.println(sb);
        sb.deleteCharAt(sb.length() - 1);
        System.out.println(sb);
        sb.reverse();
        System.out.println(sb);
        StringBuilder t = sb;
        t.append('!');
        System.out.println(sb + " " + (t == sb));
        System.out.println(new StringBuilder("ab").equals(new StringBuilder("ab")) + " "
                + new StringBuilder("ab").toString().equals("ab"));
    }
}
```

```
Java21 6
JDK Java21
JDK java21
JDK java2
2avaj KDJ
2avaj KDJ! true
false true
```

Call	Effect	Buffer
`new StringBuilder("Java")`	initial contents	Java
`append(21)`	appends at the end	Java21 (length 6)
`insert(0, "JDK ")`	inserts at position 0	JDK Java21
`setCharAt(4, 'j')`	changes the character at position 4	JDK java21
`deleteCharAt(length() - 1)`	deletes the last character	JDK java2
`reverse()`	reverses the buffer	2avaj KDJ

Table 11.14: The StringBuilder methods and their effects in the program.

The last two lines of the program show two important differences. `StringBuilder t = sb;` does not create a new buffer; `t` points to the same object. `t.append('!')` therefore changes `sb` too, and `t == sb` is true. With mutable objects, sharing leads to side effects of this kind; with `String` such a case cannot occur. The second difference is in comparison: the `equals` method of `StringBuilder` looks at the reference, not the contents. For two buffers with the same contents “ab”, `equals` gives `false`. To compare contents, `toString()` is called first.

Example 11.7

After a loop that starts with `StringBuilder sb = new StringBuilder("ab");` and does `sb.append(i).append('-')` on each iteration from 1 to 3, the lines `String s = sb.toString(); sb.reverse();` run. What does `s + " " + sb` print?

Solution. The loop turns the buffer from "ab" into "ab1-", then "ab1-2-", then "ab1-2-3-". `toString()` produces a `String` separate from the current contents: `s` = "ab1-2-3-". `reverse()` reverses only the buffer: "-3-2-1ba". `s` is not affected, because it is a separate and immutable object. In the concatenation, `toString()` is called automatically for `sb`. The program output:

```
ab1-2-3- -3-2-1ba
```

Which one is chosen when? For joining a few parts once, `+` is readable and sufficient; the Java compiler already turns one-line concatenations of this kind into efficient code. `StringBuilder` is used when text is built inside a loop or from a large number of parts whose count is not known in advance.

11.9 Applications

All the programs in this section share the same skeleton. A loop goes through the characters of the string in order or from both ends. Each character is read with `charAt` and a decision is made. What distinguishes the programs from one another is the job done at each step.

11.9.1 Reversing a string

To reverse a string, the characters are read from the end to the start and appended to a new string. The program in the slides does this in two ways: first with a hand-written loop, then with the ready-made method of `StringBuilder`.

```
String s = "Java";
String rev = "";
for (int i = s.length() - 1; i >= 0; i--)
  rev += s.charAt(i);
System.out.println(rev);
String t2 = new StringBuilder(s).reverse()
                                .toString();
System.out.println(t2);
```

```
avaJ
avaJ
```

The loop starts with `i = s.length() - 1`, that is, the last index, and decreases `i` by one as long as `i >= 0`. On each iteration `s.charAt(i)` is appended to the end of `rev`. Table 11.15 gives the four iterations.

i	i >= 0	s.charAt(i)	rev (after the iteration)
3	true	'a'	"a"
2	true	'v'	"av"
1	true	'a'	"ava"
0	true	'J'	"avaJ"
−1	false	–	the loop ends

Table 11.15: The trace table of the reversing loop. s = "Java".

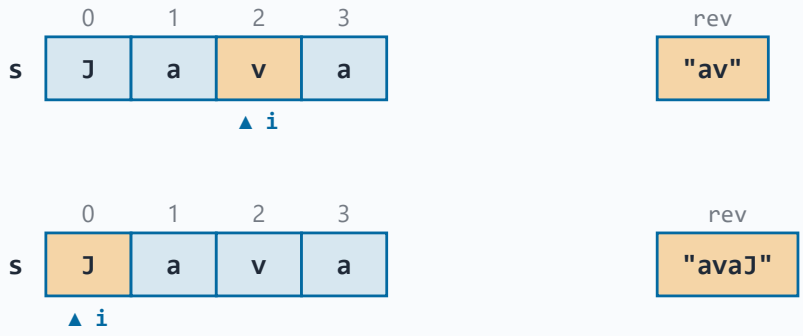


Figure 11.20: The reversing loop. Top: the i = 2 iteration, where 'v' has been appended and `rev` is "av". Bottom: the i = 0 iteration, where 'J' has also been appended and the result is "avaJ". `s` never changes.

Each `rev += s.charAt(i)` in the loop produces a new `String`; for a string of four characters this is negligible. The second way, `new StringBuilder(s).reverse().toString()`, is a single line. It creates a buffer with the contents of `s`, reverses the buffer and converts the result to a `String`. The two ways give the same output, and in both `s` does not change.

Both boundary cases of this loop work correctly. For the empty string, `i` starts at −1, the condition is false at the first check and `rev` stays empty. For a one-character string, the loop runs once and the string itself is obtained.

11.9.2 A palindrome sentence

A word or sentence that reads the same backward as forward is called a **palindrome**: “kazak”, “ana”, “Ey Edip Adanada pide ye”. Because a sentence contains uppercase letters and spaces, a direct comparison gives a wrong result: 'E' and 'e' are different characters. The text is cleaned first, then compared from both ends inward.

```
String sentence = "Ey Edip Adanada pide ye";
Locale tr = Locale.forLanguageTag("tr");
String t = sentence.toLowerCase(tr).replace(" ", "");
int left = 0, right = t.length() - 1;
boolean isPal = true;
while (left < right && isPal) {
  if (t.charAt(left) != t.charAt(right))
    isPal = false;
  left++; right--;
}
System.out.println(isPal);
```

```
true
eyedipadanadapideye 19 9 9
```

The program works in these steps:

1. `toLowerCase(tr)` lowercases all the letters with Turkish rules (`tr` is the Turkish `Locale` created on line 2), and `replace(" ", "")` removes the spaces. The 23-character sentence becomes the 19-character `"eyedipadanadapideye"`.
2. Two **pointer** variables are set up: `left` starts at 0 and `right` at the last index, 18. Until proven otherwise, `isPal = true` is assumed.
3. On each iteration the characters at the two ends are compared. If they are not equal, `isPal` is made `false` and the loop condition ends the loop at the next check. If they are equal, the two pointers move one step inward.
4. When the condition `left < right` becomes false, all the pairs have been checked.

Iteration	left	right	t.charAt(left)	t.charAt(right)	Equal?
1	0	18	e	e	yes
2	1	17	y	y	yes
3	2	16	e	e	yes
4	3	15	d	d	yes
5	4	14	i	i	yes
6	5	13	p	p	yes
7	6	12	a	a	yes

Iteration	left	right	t.charAt(left)	t.charAt(right)	Equal?
8	7	11	d	d	yes
9	8	10	a	a	yes
–	9	9	left < right is false: the loop ends, isPal = true		

Table 11.16: The trace table of the palindrome loop. t = "eyedipadanadapideye" (19 characters).

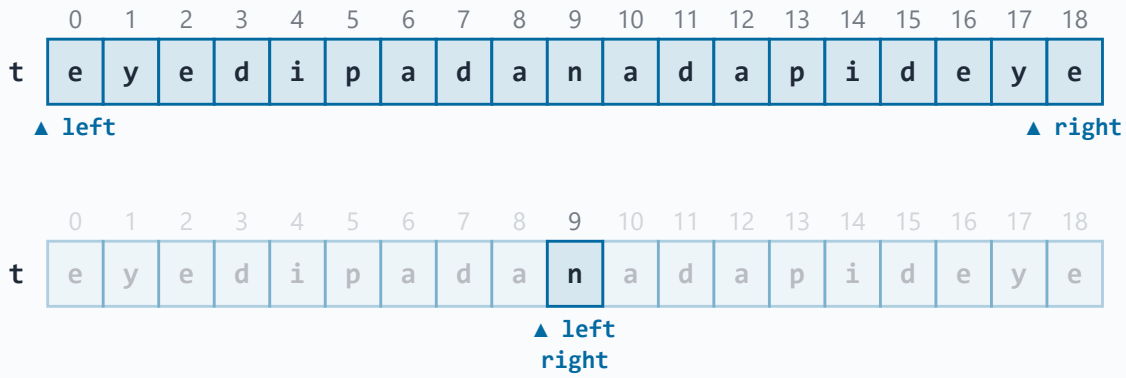


Figure 11.21: The two pointers in the palindrome check. Top: the start, with `left` at 0 and `right` at 18. Bottom: the final state, where the pointers have met at the middle index 9. The middle 'n' is not compared with itself.

The second line of the program's output is a check line showing the variables after the loop: the cleaned text, its length 19 and the final values of the pointers, 9 and 9. Because the length is odd, a character with no partner is left in the middle. Nine iterations are enough: *⌊19 / 2⌋ = 9*. The general rule is this: for a text of n characters, at most n/2 iterations are needed.

The same logic can be put into a method and tried on several sentences. In this version too the lowercasing is done with Turkish rules; this way sentences containing “I” and “İ” are processed correctly:

```
import java.util.Locale;

public class PalindromeTest {
    static boolean isPalindrome(String sentence) {
        String t = sentence.toLowerCase(Locale.forLanguageTag("tr")).replace(" ", "");
        int left = 0, right = t.length() - 1;
        boolean isPal = true;
        while (left < right && isPal) {
            if (t.charAt(left) != t.charAt(right))
                isPal = false;
            left++; right--;
        }
        return isPal;
    }

    public static void main(String[] args) {
        String[] sentences = {"Ey Edip Adanada pide ye", "Kaç kişi", "Ana", "Kazak",
"Ali"};
        for (String c : sentences) {
            System.out.println(c + " -> " + isPalindrome(c));
        }
    }
}
```

```
Ey Edip Adanada pide ye -> true
Kaç kişi -> false
Ana -> true
Kazak -> true
Ali -> false
```

“Kaç kişi” is not a palindrome: its cleaned form is “kaçkişi” and its reverse is “işikçak”. In the first iteration 'k' meets 'i' and the loop ends there. “Ali” is also eliminated in the first iteration: 'a' ≠ 'i'.

11.9.3 Counting vowels

A short way to tell whether a character belongs to a certain set is to gather the set in a string and search it with `indexOf`. If the character is in the set, `indexOf` returns 0 or a larger value; if not, −1.

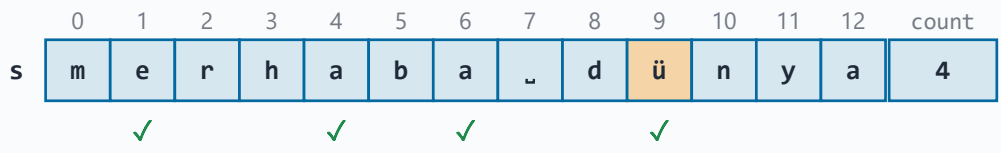
```
Locale tr = Locale.forLanguageTag("tr");
String s = "Merhaba Dünya".toLowerCase(tr);
String vowels = "aeıioöuü";
int count = 0;
for (int i = 0; i < s.length(); i++) {
  char c = s.charAt(i);
  if (vowels.indexOf(c) >= 0) count++;
}
System.out.println(count);
```

```
5
```

“Merhaba Dünya” is lowercased first: “merhaba dünya”. The string `vowels` holds the eight vowels of Turkish: a, e, ı, i, o, ö, u, ü. The loop tries the thirteen characters in turn. Table 11.17 gives the result of `indexOf` for each character.

i	c	vowels.indexOf(c)	Vowel?	count
0	m	−1	no	0
1	e	1	yes	1
2	r	−1	no	1
3	h	−1	no	1
4	a	0	yes	2
5	b	−1	no	2
6	a	0	yes	3
7	␣	−1	no	3
8	d	−1	no	3
9	ü	7	yes	4
10	n	−1	no	4
11	y	−1	no	4
12	a	0	yes	5

Table 11.17: The trace table of the vowel count. vowels = "aeıioöuü"; ␣ is a space.



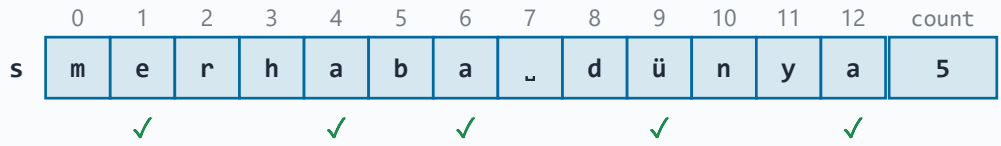


Figure 11.22: The vowel scan. Top: i = 9, where 'ü' is found at index 7 of the string `vowels` and the counter becomes 4. Bottom: the scan has finished; the five marked cells (e, a, a, ü, a) are vowels.

The `toLowerCase(tr)` call at the start of the program is important here. Without the lowercasing, uppercase letters such as "A" and "E" could not be found in the string `vowels`. The parameterless `toLowerCase()` depends on the language of the machine: if the text has an uppercase "I" or "İ", it may turn into the wrong lowercase letter on a non-Turkish machine. This is why `Locale.forLanguageTag("tr")` is given for Turkish text.

Example 11.8

What does the same program count for "Giresun Üniversitesi"? Which letters are counted?

Solution. The lowercased text is "giresun üniversitesi". The vowels are, in order, i, e, u (giresun: three), ü, i, e, i, e, i (üniversitesi: six). The total is *3 + 6 = 9*. For the space and the consonants `indexOf` returns −1 and the counter does not increase.

11.9.4 Counting words

The number of words in a sentence cannot be found by counting the spaces and adding one. There can be several spaces at the start of the sentence, at the end, or between words. A more robust way is to count the transitions "from outside a word to inside a word".

Definition 11.13

In this program, a **word** is an unbroken sequence of non-space characters. The number of words in a text equals the number of positions at which a non-space character is immediately preceded by a space or by the start of the text.

```
String text = "Ali  ata bak";
int word = 0;
boolean inWord = false;
for (int i = 0; i < text.length(); i++) {
  char c = text.charAt(i);
  if (c != ' ' && !inWord) {
    word++;
    inWord = true;
  } else if (c == ' ') inWord = false;
}
System.out.println(word);
```

```
3
```

The program keeps a `boolean` flag: `inWord` shows whether we are currently inside a word. When a non-space character arrives and the flag is `false`, a new word has begun: the counter increases and the flag is set to `true`. When a space arrives, the flag is set to `false`. If a non-space character arrives while the flag is already `true`, nothing is done; the same word continues.

i	c	inWord (before)	Event	word	inWord (after)
0	A	false	new word	1	true
1	l	true	continues	1	true
2	i	true	continues	1	true
3	␣	true	the word ends	1	false
4	␣	false	space, not counted	1	false
5	a	false	new word	2	true
6	t	true	continues	2	true
7	a	true	continues	2	true
8	␣	true	the word ends	2	false
9	b	false	new word	3	true
10	a	true	continues	3	true
11	k	true	continues	3	true

Table 11.18: The trace table of the word-counting loop. text = "Ali␣␣ata␣bak" (12 characters).

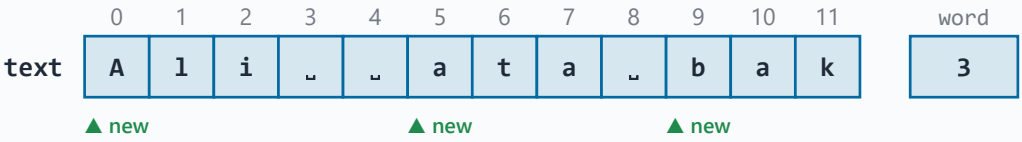


Figure 11.23: The three word starts in the text "Ali␣␣ata␣bak". The ▲ marks show the indices 0, 5 and 9, where the text goes from a space to a letter. The double space (indices 3 and 4) does not start a second word.

The same result can also be found with `split` in one line: `text.trim().split("\\s+").length`. `trim()` removes leading and trailing spaces; the pattern `"\\s+"` means "one or more whitespace characters" and counts consecutive spaces as a single delimiter. This shortcut has a trap: for an empty text or one made only of spaces, the result is 1, not 0. `"".split(...)` returns an array with one part, and that part is the empty string. The program below compares the three methods on five texts:

```
public class WordShortcut {
    public static void main(String[] args) {
        String[] texts = {"Ali  ata bak", "  Ali ata  bak ", "", "   ", "tek"};
        for (String text : texts) {
            int k1 = text.trim().split("\\s+").length;
            int k2 = text.isBlank() ? 0 : text.trim().split("\\s+").length;
            System.out.println("[" + text + "] " + k1 + " " + k2 + " " + text.split("
").length);
        }
    }
}
```

```
[Ali  ata bak] 3 3 4
[  Ali ata  bak ] 3 3 6
[] 1 0 1
[   ] 1 0 0
[tek] 1 1 1
```

Each line prints, in order, the result of the shortcut, the result with the `isBlank()` check and the result of `split(" ")`. The first two columns are the same except for the empty texts. `split(" ")`, however, counts every single space as a delimiter and so produces empty parts for double spaces: for "Ali␣␣ata␣bak" it gives 4 instead of 3. This is why the shortcut is used after checking with `isBlank()` first.

11.9.5 Character frequency

To count how many times each letter occurs in a text, an `int` array of 26 cells is used. Cell 0 is the counter for the letter 'a', cell 1 for 'b' and cell 25 for 'z'. The cell of a letter is found with `c - 'a'`, taking advantage of the fact that `char`s are numbers.

```
String s = "kavak";
int[] freq = new int[26];
for (int i = 0; i < s.length(); i++) {
  char c = s.charAt(i);
  if (c < 'a' || c > 'z') continue;
  freq[c - 'a']++;
}
for (int k = 0; k < 26; k++)
  if (freq[k] > 0)
    System.out.println((char) ('a' + k)
                       + ": " + freq[k]);
```

```
a: 2
k: 2
v: 1
```

i	c	c – 'a'	Cell updated	freq[0]	freq[10]	freq[21]
0	k	107 – 97 = 10	freq[10]	0	1	0
1	a	97 – 97 = 0	freq[0]	1	1	0
2	v	118 – 97 = 21	freq[21]	1	1	1
3	a	97 – 97 = 0	freq[0]	2	1	1
4	k	107 – 97 = 10	freq[10]	2	2	1

Table 11.19: How the frequency array is updated for “kavak”. Only the cells that change are shown.

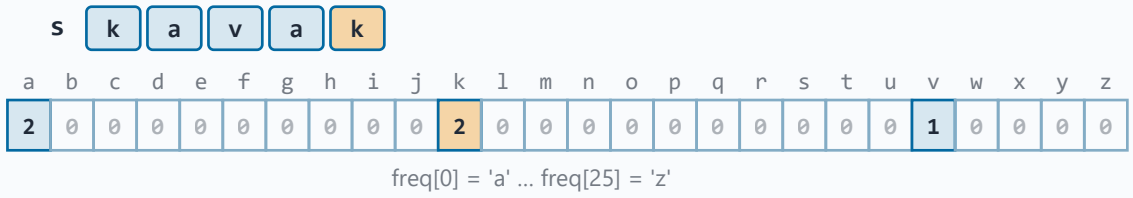


Figure 11.24: The 26-cell frequency array after “kavak” has been scanned. Only the cells for 'a' (2), 'k' (2) and 'v' (1) are nonzero.

The second loop goes through the 26 cells in order and prints only those greater than zero. To get from a cell number back to a letter, `(char) ('a' + k)` is written: *'a' + 10 = 97 + 10 = 107*, that is, 'k'. The cast (`(char)`) is needed; otherwise `'a' + k` would be an `int` and 107 would be printed. That the output is in alphabetical order also comes from this second loop: the cells are visited from 0 to 25.

The two costs of the program can be computed separately. The first loop scans a text of n characters once; the second loop always runs 26 iterations. The total work is proportional to $n + 26$. A solution that scans the text from the start for each letter would make $26 \cdot n$ comparisons.

Common mistake

Skipping the range check on line 5. The 26-cell table is only for lowercase English letters. If the program is run without the check on a text containing uppercase letters, spaces or Turkish letters, then, for example for 'ç', *'ç' − 'a' = 231 − 97 = 134* comes out and the bound of the array is exceeded:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 134 out
of bounds for length 26
```

The check skips characters outside a–z with `continue`. If uppercase letters are to be counted, they are lowercased first. For Turkish letters, either an alphabet string of 29 letters and `indexOf` are used, or a wider table is built.

11.9.6 The Caesar cipher

The **Caesar cipher** is a very old encryption method that shifts every letter forward by a fixed number of positions in the alphabet. If the shift is 3, 'a' → 'd' and 'b' → 'e'. The letters at the end of the alphabet wrap around to the start: 'x' → 'a', 'y' → 'b', 'z' → 'c'.

The arithmetic of wrapping around is the **mod** operation. The place of a letter in the alphabet is converted to a number from 0 to 25 with `c - 'a'`. The shift is added and the remainder of the result divided by 26 is taken:

$$\text{pos} = (c - \text{'a'} + \text{shift}) \bmod 26$$

The remainder is always between 0 and 25; so the result does not spill outside the alphabet. Then `'a' + pos` turns it back into a letter. The program encrypts only lowercase English letters (a–z); spaces, digits, uppercase and Turkish letters are appended unchanged by the check on line 6.

```
String text = "yazilim";
int shift = 3;
StringBuilder enc = new StringBuilder();
for (int i = 0; i < text.length(); i++) {
  char c = text.charAt(i);
  if (c < 'a' || c > 'z') { enc.append(c); continue; }
  int pos = (c - 'a' + shift) % 26;
  char out = (char) ('a' + pos);
  enc.append(out);
}
System.out.println(enc);
```

```
bdclolp
```

i	c	c – 'a'	(c – 'a' + 3) % 26	new	enc
0	y	24	27 % 26 = 1	b	b
1	a	0	3 % 26 = 3	d	bd
2	z	25	28 % 26 = 2	c	bdc
3	i	8	11 % 26 = 11	l	bdcl
4	l	11	14 % 26 = 14	o	bdclo
5	i	8	11 % 26 = 11	l	bdclol
6	m	12	15 % 26 = 15	p	bdclolp

Table 11.20: Encrypting the word “yazilim” with a shift of 3.

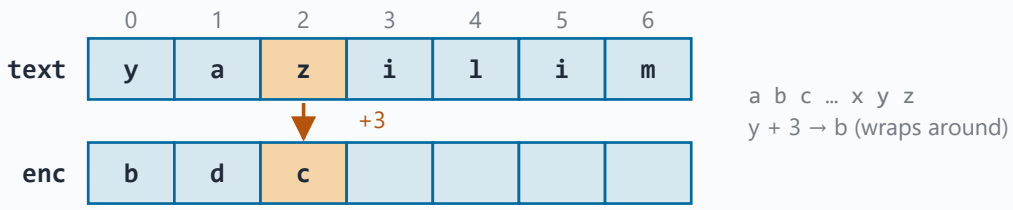


Figure 11.25: Encrypting the letter 'z'. *25 + 3 = 28* is outside the alphabet; *28 % 26 = 2* wraps around and 'c' is obtained. In the `enc` buffer “bdc” has accumulated so far.

To decrypt, the letters are shifted back by the same amount. But writing `(c - 'a' - 3) % 26` is dangerous: in Java the remainder of a negative number is negative, and *(0 – 3) % 26 = –3*. Instead of shifting back, a forward shift of *26 – 3 = 23* is made; going 23 forward in the alphabet is the same as going 3 back, and the result always stays positive. The program below puts the encryption into a method and decrypts with the same method:

```
public class CaesarDecrypt {
    static String shift(String text, int shift) {
        StringBuilder result = new StringBuilder();
        for (int i = 0; i < text.length(); i++) {
            char c = text.charAt(i);
            if (c < 'a' || c > 'z') { result.append(c); continue; }
            int pos = (c - 'a' + shift) % 26;
            result.append((char) ('a' + pos));
        }
        return result.toString();
    }

    public static void main(String[] args) {
        String enc = shift("yazilim", 3);
        System.out.println(enc);
        System.out.println(shift(enc, 26 - 3));
        System.out.println(shift("kaz", 2));
        System.out.println((-3) % 26);
    }
}
```

```
bdclolp
yazilim
mcb
-3
```

The second line is the decrypted text: “yazilim”. The third line is the encryption of the word “kaz” with a shift of 2: 'k' (10) → 12 → 'm', 'a' → 'c', 'z' (25) → *27 % 26 = 1* → 'b'. The last line confirms the value of a negative remainder in Java. The Caesar cipher has only 25 meaningful keys; a computer tries them all at once. This is why the method is used not for security but to teach modular arithmetic and character processing.

11.9.7 Worked examples from the exercises

Six of the exercises in the slides are solved below. Each solution uses one or more of the tools described in the notes.

Example 11.9

A program that finds separately the numbers of uppercase letters, lowercase letters, digits and spaces in a sentence.

Solution. Each character is classified with an else-if chain. The methods `isUpperCase`, `isLowerCase` and `isDigit` of the `Character` class also recognize Turkish letters; so “Ö” counts as uppercase and “ı” as lowercase.

```
public class CharTypes {
    public static void main(String[] args) {
        String sentence = "Giresun 2024 Yılı Öğrenci";
        int upper = 0, lower = 0, digit = 0, space = 0, other = 0;
        for (int i = 0; i < sentence.length(); i++) {
            char c = sentence.charAt(i);
            if (Character.isUpperCase(c)) upper++;
            else if (Character.isLowerCase(c)) lower++;
            else if (Character.isDigit(c)) digit++;
            else if (c == ' ') space++;
            else other++;
        }
        System.out.println("Upper: " + upper + ", lower: " + lower + ", digit: " +
digit
                + ", space: " + space + ", other: " + other);
    }
}
```

```
Upper: 3, lower: 15, digit: 4, space: 3, other: 0
```

In the sentence “Giresun 2024 Yılı Öğrenci”, G, Y and Ö are uppercase. The lowercase letters are “iresun” (6), “ılı” (3) and “ğrenci” (6): 15 in total. There are four digits and three spaces. The total *3 + 15 + 4 + 3 = 25* equals the length of the sentence.

Example 11.10

A program that checks with character frequencies whether two words are anagrams (a rearrangement of the same letters, e.g. “elma” – “alem”).

Solution. If the lengths differ, the words cannot be anagrams. If the lengths are the same, a single frequency array is enough: the letters of the first word increase the counters and those of the second decrease them. If all the cells are zero in the end, the two words have the same letters in the same numbers.

```
public class Anagram {
    static boolean isAnagram(String a, String b) {
        if (a.length() != b.length()) return false;
        int[] freq = new int[26];
        for (int i = 0; i < a.length(); i++) {
            freq[a.charAt(i) - 'a']++;
            freq[b.charAt(i) - 'a']--;
        }
        for (int k = 0; k < 26; k++)
            if (freq[k] != 0) return false;
        return true;
    }

    public static void main(String[] args) {
        System.out.println(isAnagram("elma", "alem"));
        System.out.println(isAnagram("kavak", "kavka"));
        System.out.println(isAnagram("kalem", "kelam"));
        System.out.println(isAnagram("kitap", "katip"));
        System.out.println(isAnagram("ders", "dere"));
    }
}
```

```
true
true
true
true
false
```

“ders” and “dere” have the same length, but the 's' cell is left at +1 and the 'e' cell at −1; the result is `false`. The program is written only for lowercase English letters; the warning in the frequency section applies here too for Turkish letters.

Example 11.11

A program that prints the full name entered by the user in the form “SURNAME, First”, with the surname in uppercase.

Solution. The line is read with `nextLine()`, because the full name contains spaces. For people with several first names, the surname is taken to be the last word and is separated with `lastIndexOf(' ')`. The conversion to uppercase is done with Turkish rules.

```
import java.util.Locale;
import java.util.Scanner;

public class FullName {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String line = in.nextLine().strip();
        int space = line.lastIndexOf(' ');
        String first = line.substring(0, space).strip();
        String last = line.substring(space + 1);
        System.out.println(last.toUpperCase(Locale.forLanguageTag("tr")) + ", " +
first);
    }
}
```

When “Ada Nur Yılmaz” is entered, the output is:

```
YILMAZ, Ada Nur
```

When “␣␣Mehmet Işık␣” is entered, `strip()` removes the margin spaces and the output is:

```
IŞIK, Mehmet
```

The uppercase form of “Işık” under Turkish rules is “IŞIK”. The program fails on a single-word input because `lastIndexOf` returns −1; this case can be checked separately.

Example 11.12

A program that finds the longest word in a sentence with `split`.

Solution. The sentence is split into words with the pattern `"\\s+"`. The first candidate is the first word; each word is compared with the candidate and, if longer, becomes the new candidate.

```
public class LongestWord {
    public static void main(String[] args) {
        String sentence = "Bilgisayar mühendisliği  öğrencileri dizgileri seviyor";
        String[] words = sentence.trim().split("\\s+");
        String longest = words[0];
        for (String k : words) {
            if (k.length() > longest.length()) longest = k;
        }
        System.out.println(longest + " " + longest.length() + " " + words.length);
    }
}
```

```
mühendisliği 12 5
```

Although the sentence has a double space, five words are found. “mühendisliği” is the longest with 12 characters. “öğrencileri” has 11 characters. Because the condition is `>`, if two words have the same length the one that comes first is kept.

Example 11.13

A program that extends the Caesar cipher to support uppercase letters (A–Z) too and leaves other characters as they are.

Solution. Each character falls into one of three groups. Lowercase letters are shifted relative to 'a' and uppercase letters relative to 'A'. Digits, spaces and punctuation are appended unchanged.

```
public class CaesarUpper {
    static String encrypt(String text, int shift) {
        StringBuilder result = new StringBuilder();
        for (int i = 0; i < text.length(); i++) {
            char c = text.charAt(i);
            if (c >= 'a' && c <= 'z') {
                result.append((char) ('a' + (c - 'a' + shift) % 26));
            } else if (c >= 'A' && c <= 'Z') {
                result.append((char) ('A' + (c - 'A' + shift) % 26));
            } else {
                result.append(c);
            }
        }
        return result.toString();
    }

    public static void main(String[] args) {
        String s = encrypt("Merhaba, Dunya XYZ!", 3);
        System.out.println(s);
        System.out.println(encrypt(s, 26 - 3));
    }
}
```

```
Phukded, Gxqbd ABC!
Merhaba, Dunya XYZ!
```

'M' → 'P', 'X' → 'A', 'Y' → 'B', 'Z' → 'C'; the comma, space and exclamation mark stay as they are. The second line shows that a shift of 23 turns the text back. In this program Turkish letters fall into the “other characters” group and are not encrypted; this is why “Dunya” is used in the sample text instead of “Dünya”.

Example 11.14

A program that deletes all the spaces in a string with `StringBuilder`, without using `replace`.

Solution. The string is scanned and every non-space character is appended to the buffer. Spaces are skipped.

```
public class RemoveSpaces {
    public static void main(String[] args) {
        String s = " Ey Edip  Adanada pide ye ";
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (c != ' ') sb.append(c);
        }
        System.out.println("[" + sb + "] " + sb.length() + " " + s.length());
    }
}
```

```
[EyEdipAdanadapideye] 19 26
```

The 26-character text has seven spaces: the one at the start, the one at the end, the double space after “Edip” and three single spaces. 19 characters remain. This number is the same as the length of the cleaned text in the palindrome example; apart from letter case, the result is the same too.

11.10 Formatting with printf

When a report or a table is printed, we want the values to stay aligned and decimal numbers to be cut at a certain number of digits. Concatenation with `+` does not provide this easily. The `System.out.printf` method, however, **formats** text according to a pattern.

Definition 11.14

The first parameter of `printf` is the **format string**. Plain text in the format string is written as it is. Each **format specifier** that begins with `%` stands in for the next argument and says how it is to be written. The general structure of a specifier is `%[-][width][.precision]type`. The main types are `s` (text), `d` (integer), `f` (floating-point number) and `n` (line end).

```
String name = "Java"; int qty = 3; double price = 17.5;
System.out.printf("%-8s %3d %7.2f%n", name, qty, price);
```

```
Java       3   17.50
[Java    ][  3][  17.50]
[    Java][3  ][17.5][1.750000e+01]
[Giresun][12345][3][4]
true c 50%
```

The first `printf` line of the program is the example from the slides. The three specifiers are matched with the three arguments in order. Figure 11.26 shows how the output is built cell by cell.

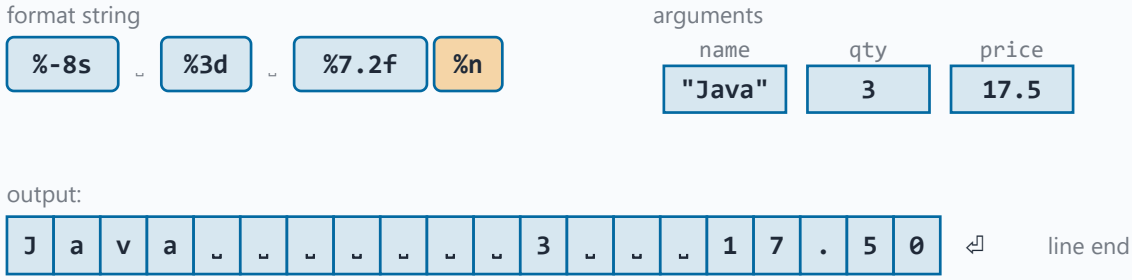


Figure 11.26: The format string `"%-8s %3d %7.2f%n"`. "Java" is left-aligned in 8 cells and completed with four spaces. Then comes the plain space in the format string. The number 3 is right-aligned in 3 cells. 17.5 becomes "17.50" with two decimals and is right-aligned in 7 cells. `%n` ends the line.

- `%-8s`: the text is written in a field of at least 8 characters. The minus sign left-aligns it; since "Java" has four characters, four spaces are added to its right.
- `%3d`: the integer is right-aligned in a field of 3 characters; two spaces come before the 3.
- `%7.2f`: the number is written with two digits after the decimal point ("17.50", five characters) and right-aligned in a field of 7 characters; two spaces come before it.
- `%n`: adds the line end appropriate to the operating system.

The following lines of the program make the limits of the fields visible with square brackets and try other formats. `%8s` right-aligns because there is no minus sign. `%-3d` left-aligns. `%.1f` writes one decimal without giving a width. `%e` is scientific notation: 1.75×10^{1}. The fourth line shows two important rules. First, the width is a lower bound: since "Giresun" has 7 characters, `%3s` does not cut it and writes all of it. Second, `%.0f` rounds halves up: 2.5 → 3 and 3.5 → 4. On the last line you can see that `%s` also writes `boolean` and `char` values. `%%` is the percent sign itself.

Format specifiers are ideal for tables. When fields of the same width are written one below the other, the columns line up on their own:

```
public class Table {
    public static void main(String[] args) {
        String[] product = {"Pen", "Notebook", "Calculator"};
        int[] qty = {12, 3, 1};
        double[] price = {7.5, 42.0, 349.9};
        System.out.printf("%-15s %5s %9s%n", "Product", "Qty", "Amount");
        for (int i = 0; i < product.length; i++) {
            System.out.printf("%-15s %5d %9.2f%n", product[i], qty[i], qty[i] * price[i]);
        }
    }
}
```

```
Product           Qty    Amount
Pen                12     90.00
Notebook            3    126.00
Calculator          1    349.90
```

Because the header line and the data lines use the same widths (15, 5, 9), the columns stand in neat order. The text column is left-aligned and the number columns right-aligned; this is the usual arrangement in tables. In the amount column, *12 × 7.5 = 90*, *3 × 42 = 126* and *1 × 349.9 = 349.9* are written with two decimals.

The decimal separator depends on the locale. The same program writes the following on a machine with a Turkish setting:

```
Product           Qty    Amount
Pen                12     90,00
Notebook            3    126,00
Calculator          1    349,90
```

The number itself does not change; only the way it is written does. The example in the slides also writes “17,50” on a Turkish machine. If the same output is wanted on every machine, the locale is given in the first parameter, as in `printf(Locale.US, …)`.

The compiler does not check the format string. If a specifier and the type of its argument do not match, or an argument is missing, the error appears while the program runs:

```
public class FormatError {
    public static void main(String[] args) {
        System.out.printf("%d%n", 3.5);
    }
}
```

```
Exception in thread "main" java.util.IllegalFormatConversionException: d !=
java.lang.Double
```

`%d` expects an integer, but `3.5` is a `double`. The message says that the `d` conversion cannot be used with `Double`. Java does not round or cut the number; it throws an exception. A similar exception occurs for a missing argument; the program stops after the first specifier has been written:

```
public class FormatMissing {
    public static void main(String[] args) {
        System.out.printf("%s and %s%n", "Ada");
    }
}
```

```
Ada and
Exception in thread "main" java.util.MissingFormatArgumentException: Format specifier '%s'
```

Common mistake

Printing a decimal value with `%d` or an integer with `%f`. Both throw an `IllegalFormatConversionException`. `%d` is used for `int` and `%f` for `double`; the safe choice for any type is `%s`, but it too provides no format checking. Forgetting the line end is also common: `printf` does not move to the next line by itself like `println`; `%n` is needed.

11.11 Unicode and text length

Java's `char` type is 16 bits and can take 65,536 different values. Almost all the characters of everyday texts, Turkish letters included, are in this range. But the Unicode standard today defines more than 150 thousand characters: old writing systems, mathematical symbols, emojis. Some of these do not fit in a single `char`.

Definition 11.15

In Unicode every character has a number; this number is called a **code point**. Java strings store these numbers with the UTF-16 encoding, in 16-bit **code units**. A `char` is one code unit. Code points greater than U+FFFF are stored as a pair made up of two code units. `length()` gives the number of code units, not the number of characters.

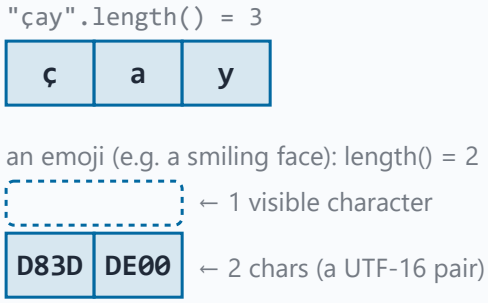


Figure 11.27: "çay" consists of three code units and its length is 3; the ç fits in a single `char`. The smiling-face emoji is a single visible character but is stored as a UTF-16 pair made up of two code units named D83D and DE00; its length is 2.

```
public class UnicodeDemo {
    public static void main(String[] args) {
        String tea = "çay";
        String emoji = "😜";
        System.out.println(tea.length() + " " + emoji.length() + " " +
emoji.codePointCount(0, emoji.length()));
        System.out.printf("%X %X %X%n", (int) emoji.charAt(0), (int) emoji.charAt(1),
emoji.codePointAt(0));
        String combined = "çay";
        System.out.println(combined.length() + " " + combined.equals(tea));
        System.out.println(java.text.Normalizer.normalize(combined,
java.text.Normalizer.Form.NFC).equals(tea));
        System.out.println("straße".toUpperCase(java.util.Locale.ROOT) + " " +
"straße".length() + " " + "straße".toUpperCase(java.util.Locale.ROOT).length());
    }
}
```

```
3 2 1
D83D DE00 1F600
4 false
true
STRASSE 6 7
```

In the first line the length of "çay" is 3 and the length of the emoji is 2. `codePointCount`, however, counts code points and gives 1 for the emoji. The second line prints the two code units of the emoji and its real code point in hexadecimal: U+1F600. Because this number is greater than 65,535, it does not fit in a single `char`.

The third line shows a subtler case. The letter "ç" can be written in two ways in Unicode: as a single ready-made character (U+00E7) or as a "c" plus a separate cedilla mark added beneath it (U+0327). The second form looks the same on the screen but has length 4 and is not equal to the first form under `equals`. The `java.text.Normalizer` class brings the two forms to the same representation;

the fourth line shows that equality holds after normalization. On the last line, when the German letter "ß" is converted to uppercase it turns into two letters, "SS", and the string grows longer: the 6-character "straße" becomes the 7-character "STRASSE".

The consequence of these details for everyday programs is this: in Turkish texts `length()` and `charAt` work as expected. In texts containing emojis, combining marks or other writing systems, however, the "number of characters the user sees" can differ from `length()`. In such texts, operations such as reversing can also split a pair in the middle and produce broken characters.

11.12 Summary

- `char` holds a single character in single quotes (`'T'`), and `String` holds a sequence of characters in double quotes (`"Tarkan"`). `String` is not a primitive type but a class in the `java.lang` package; the variable holds the **reference** to the object. `""` is a real object of length 0; `null` points to no object, and if a method is called through it, a `NullPointerException` occurs.
- `+` is **concatenation** if one of the operands is a string and addition if both are numbers, and it is evaluated from left to right: `"12" + 3 + 4` = "1234", `3 + 4 + "12"` = "712". A `char` is a number: *'a' + 1 = 98*, *'k' − 'a' = 10*.
- `String` is **immutable**. `toUpperCase` , `replace` , `substring` and `+` return a new object; if the result is not assigned, it is lost. What changes is not the object but the place the variable points to. Objects that nobody points to are cleaned up by the garbage collector.
- Indices run from 0 to `length() - 1` ; an out-of-bounds index gives a `StringIndexOutOfBoundsException` . `length()` is a method for strings, while `length` is a field for arrays.
- `substring(a, b)` : a included, b excluded; length b − a; valid range *0 ≤ a ≤ b ≤ length()*. `indexOf` gives the first position or −1. `split` takes a regular expression; consecutive delimiters produce empty parts, and trailing empty parts are dropped.
- `replace` searches for plain text and `replaceAll` for a regular expression. `trim` removes only ASCII whitespace, while `strip` removes all Unicode whitespace. `format` and `printf` depend on the locale.
- `==` compares references and `equals` compares contents; strings are always compared with `equals` . `compareTo` gives the code difference of the first differing character, or the length difference if there is none. Uppercase letters come before lowercase ones and English letters before Turkish ones; for Turkish sorting, `Collator` is used.
- In Turkish, i ↔ İ and ı ↔ I are paired. The parameterless `toLowerCase()` / `toUpperCase()` depend on the language of the machine; `Locale.ENGLISH` is given for technical text and `Locale.forLanguageTag("tr")` for Turkish text.
- The same constants share a single object in the **string pool**; `new` always creates a new object. Constant expressions are concatenated at compile time and enter the pool; a concatenation with a variable produces a new object. `intern()` gives the object in the pool. `"constant".equals(x)` is

safe against null.

- `next()` reads a word and leaves the ⏎ in the buffer; `nextLine()` reads the rest of the line and consumes the ⏎. If a line is to be read after `next()` or `nextInt()`, a `nextLine()` is put in between.
- In a loop, `s = s + x` produces a new object on every iteration; in n iterations about $n^2/2$ characters are copied. `StringBuilder` is a mutable buffer: `append`, `insert` and `reverse` change the same object; the result is obtained with `toString()`.
- The common pattern of the applications is a scan of characters. Reversing scans from the end to the start, and the palindrome check scans from both ends inward. Word counting uses a flag. A frequency table selects the cell with `c - 'a'`; the Caesar cipher shifts with `(c - 'a' + k) % 26`.
- In `printf`, `%-8s` left-aligns and `%3d` right-aligns; `%7.2f` writes with two decimals in 7 cells; `%n` is the line end. A type mismatch gives an exception at run time.
- Strings are made up of UTF-16 **code units**; Turkish letters fit in a single `char`, while characters such as emojis take two `char`s. So `length()` is not always the same as the number of visible characters.

11.13 Key terms

These are the key terms introduced in this chapter.

string character string literal primitive type object reference heap string concatenation immutable immutability garbage collector index indexing out of bounds substring delimiter regular expression whitespace content vs reference comparison lexicographic order uppercase / lowercase locale string pool buffer mutable reversing Caesar cipher formatting format string format specifier code point code unit exception

---

11.14 Test Yourself

1. With `int p = 2, q = 5;`, what does the line `System.out.println(p + q + "x" + p + q);` print?

   Answer

   `+` is evaluated from left to right. The first two operands are numbers: *2 + 5 = 7*. Then *7 + "x"* = "7x", and the expression turns into concatenation: "7x2", then "7x25". The output is `7x25`.

2. After the lines `String s = "dizgi"; s.toUpperCase(); s = s.replace('i', 'ı');`, what does

`s` point to?

Answer

The result of `toUpperCase()` is lost because it is not assigned; `s` is still "dizgi". The result of `replace`, however, is assigned and both 'i's become 'ı': `s` points to "dızgı". The "dizgi" object has not changed; only the arrow of `s` has moved to the new object.

3. For `String t = "Karadeniz";`, what are the values of `t.length()`, `t.charAt(4)`, `t.indexOf("de")`, `t.substring(4, 8)` and `t.lastIndexOf('a')`?

Answer

The indices: K=0, a=1, r=2, a=3, d=4, e=5, n=6, i=7, z=8. The length is 9. `charAt(4)` is 'd'. "de" starts at index 4. `substring(4, 8)` takes indices 4–7: "deni" (length $8 - 4 = 4$). The last 'a' is at index 3.

4. What is `"x;;y;z;;".split(";").length`? What are the parts?

Answer

The text is cut at every semicolon: "x", "", "y", "z", "", "". The empty part between two consecutive semicolons stays; the two trailing empty parts are dropped. The result is `[x, , y, z]`, with length 4.

5. What are the values of `"Masa".compareTo("masa")`, `"kitap".compareTo("kitaplık")` and `"ödev".compareTo("sınav")`? Which of them contradict true dictionary order?

Answer

In the first, the first characters are 'M' (77) and 'm' (109): $77 - 109 = -32$. In the second, "kitap" is the beginning of "kitaplık": $5 - 8 = -3$. In the third, 'ö' (246) meets 's' (115): $246 - 115 = 131$, so "ödev" comes after "sınav". Because ö comes before s in the Turkish alphabet, the third result contradicts dictionary order.

6. On a computer with a Turkish setting, what is the value of the expression `"MAIL".toLowerCase().equals("mail")`? How is the expression fixed?

Answer

The parameterless `toLowerCase()` uses Turkish rules and converts "I" to "ı": "maıl". Because it is not equal to "mail", the value is `false`. Since "MAIL" is a technical word, `toLowerCase(Locale.ENGLISH)` is written, or `"MAIL".equalsIgnoreCase("mail")` is used directly; both give `true`.

7. After the lines `String a = "ab"; String b = "a"; String c = b + "b"; final String d = "a"; String e = d + "b";`, what are the values of `a == c`, `a == e` and `a == c.intern()`?

   Answer

   Because `b` is not `final`, `b + "b"` produces a new object at run time: `a == c` is `false`. Because `d` is a `final` constant, `d + "b"` becomes the constant "ab" at compile time and comes from the pool: `a == e` is `true`. `c.intern()` returns the "ab" in the pool: `true`.

8. The user types "42⏎" first, then "Ali Can⏎". After the lines `int n = in.nextInt(); String s = in.nextLine(); String t = in.next();`, what are `s` and `t`?

   Answer

   `nextInt()` reads 42 and leaves the ⏎. `nextLine()` reads up to that ⏎: `s` is the empty string. `next()` reads the first word of the second line: `t` is "Ali". " Can⏎" stays in the buffer.

9. The word "ankara" is encrypted with a Caesar cipher with a shift of 4. What is the result? What shift is used to decrypt?

   Answer

   The places of the letters: a=0, n=13, k=10, r=17. Adding 4 gives 4, 17, 14, 21; none exceeds 26. The six letters turn, in order, into e, r, o, e, v, e: "eroeve". To decrypt, a shift of *26 − 4 = 22* is used.

10. What does the line `System.out.printf("[%6.1f][%-4s][%03d]%n", 2.75, "ab", 7);` print?

    Answer

    `%6.1f` rounds 2.75 to one decimal: "2.8" (three characters) and right-aligns it in 6 cells; three spaces come before it. `%-4s` left-aligns "ab" in 4 cells; two spaces come after it. `%03d` pads 7 with zeros to 3 cells: "007". The output is `[ 2.8][ab ][007]` (with the English locale; "2,8" with the Turkish one).

11.15 Further reading

- Oracle, *The Java Tutorials*, the pages "Strings" and "The StringBuilder Class": creating strings, concatenation, methods and `StringBuilder` are explained with official examples.
- Oracle, *Java SE 21 API Documentation*, the classes `java.lang.String`, `java.util.Formatter` and `java.util.Locale`: the exact behavior of each method, the empty-part rules of `split` and the full list of format specifiers.

- R. Sedgewick and K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), Section 3.1 “Using Data Types”: `String` is treated as a data type; search examples on genome strings are given.
- Unicode Consortium, *The Unicode Standard*, the “Case Mappings” section and the file `SpecialCasing.txt`: the source of the Turkish and Azeri i/İ/ı/I rules and of length-changing conversions such as “ß → SS”.

11.16 References

1. P. Deitel, H. Deitel, *Java: How to Program, Early Objects*, 11th ed., Pearson, 2018.
2. Y. D. Liang, *Introduction to Java Programming and Data Structures*, 13th ed., Pearson, 2024.
3. Oracle, *The Java Tutorials: Strings; The StringBuilder Class*.
4. Oracle, *Java SE 21 API: String, StringBuilder*.
5. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 3.1 Using Data Types.
6. Unicode Consortium, *The Unicode Standard*.

Recursion

Some problems contain a smaller copy of themselves. 4! is 4 times 3!, and 3! is 3 times 2!. A method that solves such a problem can hand the smaller part of the work back to itself. A method that calls itself is called recursive, and the technique is called recursion. These notes first introduce the two parts every recursive method needs, the base case and the recursive case. They then follow, step by step, how pending calls pile up in memory, which is the call stack. After examples such as sums, digit sums, palindromes, the greatest common divisor, exponentiation, binary search, merge sort, the Towers of Hanoi and subset generation, the notes turn to cost. Numbers show when recursion is more expensive than a loop, why the Fibonacci example repeats the same work thousands of times, and how memoization removes that repetition. Every Java program was compiled and run with JDK 21, and the outputs, call counts and stack depths in the notes come from those runs.

LEARNING OUTCOMES

Structure
Identify the base case and the recursive case of a recursive method; explain what measure shrinks with each call and why the base case is always reached.

Tracing
Trace a recursive call step by step on the call stack; tell which work happens on the way down and which on the way back; find the number of frames on the stack at its deepest point.

Comparison
Compare a recursive solution with a loop in time, extra memory and readability; explain the causes of stack overflow and integer overflow.

Improvement
Find repeated subproblems in a recursion tree; use memoization to reduce the number of calls from exponential to linear; explain why tail recursion does not shrink the stack in Java.

12.1 What is recursion?

12.1.1 A method that calls itself

Methods can call one another. `main` calls a method, and that method may call another. When the called method finishes, control returns to the caller and the program continues where it left off. In Java, the methods a method may call include the method itself: a method can use its own name inside its own body.

At first this looks odd: a method that calls itself seems as if it could never finish. The picture changes if every call is made on a slightly smaller version of the problem. As the problem shrinks, it eventually becomes small enough that the answer is known directly. At that point the method does not call itself again; it returns the answer immediately. The waiting calls then use that answer, one after another, to finish their own work.

Definition 12.1

Recursion is the technique in which a method reduces a problem to a smaller instance of the same problem and calls itself to solve it. A method that calls itself is a **recursive method**.

Factorial illustrates the idea well. In mathematics, $4! = 4 \times 3 \times 2 \times 1 = 24$. The same product can be written $4! = 4 \times 3!$, so to find 4! it is enough to find 3! first. Computing 3! needs 2!, and 2! needs 1!. The value 1! is known directly: it is 1. Figure 12.1 shows this nesting. Each box contains a smaller copy of the one before it, and the innermost box does not open another box inside itself.

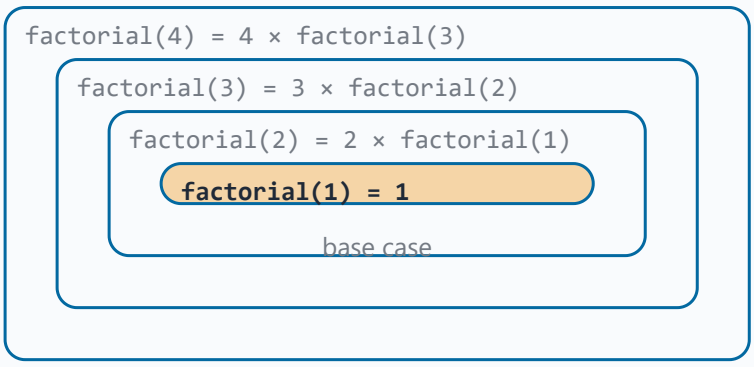


Figure 12.1: Nested calls. factorial(4) calls factorial(3), which calls factorial(2). The innermost call, factorial(1), is the base case; it returns 1 without making a new call.

Each call works with its own parameters and its own local variables. The `n` in the factorial(4) call and the `n` in the factorial(3) call have the same name but are separate variables: one holds 4, the other 3. Java reserves a separate area of memory for every call. This area is called a **stack frame**, and

frames are stacked on top of one another in the order the calls were made. The structure formed by the frames is the **call stack**. The call stack is the key to understanding recursion, and the notes rely on it throughout.

Definition 12.2

A **stack frame** is the area of memory set up for one method call. It holds the call's parameters, its local variables and the place to return to when it finishes. The frames of all the calls currently running sit on top of one another in the **call stack**. The frame of a new call is added on top; when the call finishes, the top frame is removed.

12.1.2 The two required parts

Every correct recursive method has two parts. One is the case in which the answer is known directly. The other is the case in which the problem is reduced and the method calls itself again.

Definition 12.3

The **base case** is the case in which the result is known directly and no new call is made. For factorial, the result is 1 when $n \le 1$. The **recursive case** is the case in which the problem is turned into a smaller instance and the method calls itself. For factorial, this is the computation $n \times factorial(n - 1)$.

Figure 12.2 shows the order of these two parts within a call. When a call starts, it first asks whether this is the base case. If the answer is yes, the result is returned at once. If it is no, the problem is reduced and a new call is made. The new call starts with the same question.

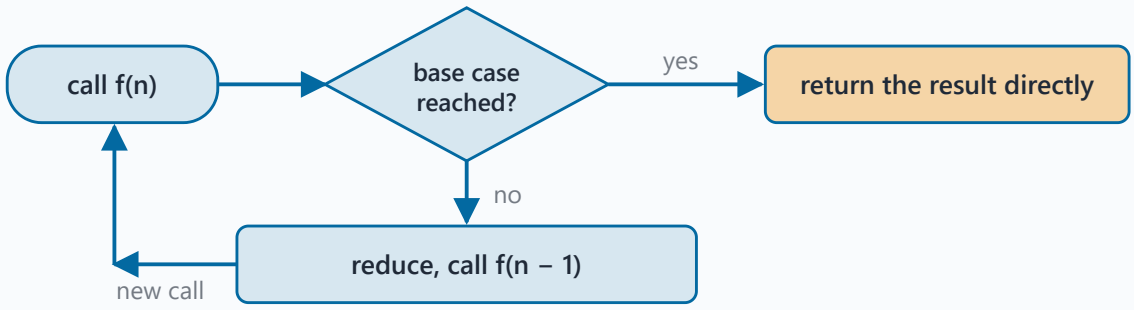


Figure 12.2: The flow of a recursive call. At the base case, the result is returned directly. Otherwise the problem is reduced and f(n − 1) is called, and that new call enters the flow from the start.

If either part is missing, the method does not work correctly. Without a base case the calls never stop: every call opens a new frame, and sooner or later memory runs out. With a base case but a problem that does not get closer to it on every call, the outcome is the same. In the line `return n * factorial(n);`, for example, n never shrinks and the base case is never reached.

For this reason, three questions are asked when writing a recursive method:

1. **What is the base case?** Which is the smallest input whose answer is known without computation? For factorial it is $n \leq 1$.
2. **How does the problem shrink?** Which measure decreases on every call? For factorial, n decreases by 1 on every call.
3. **How does the small answer become the big answer?** What is done with the result of the subcall? For factorial, the subresult is multiplied by n.

The answer to the second question is also the reason the method terminates. Here n is an integer and decreases by 1 on every call. Starting from 4, n reaches 1 in at most three steps. A decreasing integer reaches a lower bound in a finite number of steps, and this is how we show that the base case can be reached.

> **Common mistake**
>
> Writing the base case so that it skips some inputs. The method `if (n == 0) return 0; return n + f(n - 2);` works for even n. In the call f(5), however, n takes the values 5, 3, 1, −1, −3, and so on. None of the odd numbers equals 0, so the base case is skipped and the calls continue. Changing the condition to `n <= 0` fixes this. When the input shrinks in jumps, a "less than or equal" test is safer than a test for equality.

12.1.3 Thinking recursively

When writing a recursive method, there is no need to picture all the calls at once. Instead, think this way: "The method already works correctly for every input smaller than this one. Given that assumption, how do I build the answer for the input I was given?" For factorial the answer is short. If factorial(n − 1) gives the correct result, then n! is n times that result.

This way of thinking resembles induction in mathematics. In induction, the claim is first shown for the smallest case. Then, assuming the claim holds for k, it is shown to hold for k + 1. The base case corresponds to the starting step of induction, and the recursive case to the inductive step. If both steps are correct, the method returns the correct result for every valid input.

Even so, the safest way for a beginner is to trace the method by hand on a small input. Which call is opened when, what it waits for and what it returns are written in a table. In the rest of these notes nearly every example is traced with such a table. All the tables were produced on JDK 21 by adding lines to the methods that record the moments of call and return.

12.2 Factorial and the call stack

12.2.1 Recursive factorial

The mathematical definition of factorial takes two lines: *0! = 1! = 1* and *n! = n × (n − 1)!*. The factorial of a negative number is not defined. The Java method follows this definition almost word for word:

```
static long factorial(int n) {
    if (n < 0)
        throw new IllegalArgumentException();
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

The method works line by line as follows:

1. `static long factorial(int n)` : the method takes an `int` and returns a `long` . Factorial grows very fast, so the result type is `long` . An int can hold at most 12!; a long is enough up to 20!.
2. `if (n < 0) throw new IllegalArgumentException();` : a negative input is invalid. In this case the method throws an exception object and stops running. Throwing is Java's way for a method to say “I cannot do the job with this input.”
3. `if (n <= 1) return 1;` : the base case. For 0 and 1 the result is 1, and no new call is made.
4. `return n * factorial(n - 1);` : the recursive case. First factorial(n − 1) is called. When that call finishes and returns a value, the value is multiplied by n and the product is returned.

The order in the last line matters. The multiplication cannot be done until the result of the subcall arrives, so factorial(4) leaves its multiplication half finished and waits. The factorial(3) call waits in the same way. All the multiplications happen on the way back.

Figure 12.3 shows, on a number line, what the method does depending on the region the input falls in. Negative numbers are in the error region. 0 and 1 are the base case. From 2 upward lies the recursive region; each call moves one step to the left and finally lands in the base case.

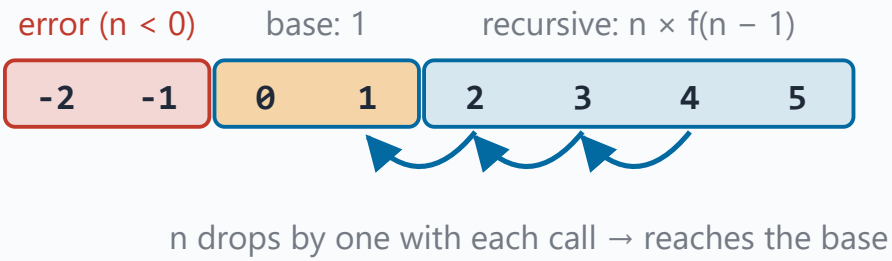


Figure 12.3: The input regions of the factorial method. n < 0 is an error, n = 0 and n = 1 are the base case, and n ≥ 2 is the recursive case. In the call factorial(4), n goes down through 4, 3, 2, 1.

Run on a few inputs, the method gives the expected results. The first line of the output below shows factorial(4), factorial(0), factorial(1) and factorial(20). The second line shows the exception thrown by the call factorial(−3), caught and printed:

```
24 1 1 2432902008176640000
factorial(-3): java.lang.IllegalArgumentException
```

12.2.2 Tracing factorial(4) step by step

The trace slide in the presentation uses the program below. The `main` method calls factorial(4) and prints the result:

```
static long factorial(int n) {
    if (n < 0)
        throw new IllegalArgumentException();
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
System.out.println(factorial(4));
```

```
24
```

Table 12.1 gives, step by step, what happens on the call stack while the program runs. The “Frames” column is the number of frames on the stack at that moment, and it includes the frame of `main`.

Step	Event	Frames (including main)	Stack (bottom to top)	Explanation
1	factorial(4) called	2	main, f(4)	waits for 4 × factorial(3)
2	factorial(3) called	3	main, f(4), f(3)	waits for 3 × factorial(2)

Step	Event	Frames (including main)	Stack (bottom to top)	Explanation
3	factorial(2) called	4	main, f(4), f(3), f(2)	waits for 2 × factorial(1)
4	factorial(1) called	5	main, f(4), f(3), f(2), f(1)	base case
5	factorial(1) returned	5	main, f(4), f(3), f(2), f(1)	return 1
6	factorial(2) returned	4	main, f(4), f(3), f(2)	2 × 1 = 2
7	factorial(3) returned	3	main, f(4), f(3)	3 × 2 = 6
8	factorial(4) returned	2	main, f(4)	4 × 6 = 24
9	main prints the result	1	main	24

Table 12.1: The state of the stack in the call factorial(4). f(k) is the frame of the call factorial(k).

The first four rows of the table are the descent phase. factorial(4) is called and a frame is opened. In this frame n = 4. Since $4 < 0$ and $4 \leq 1$ are both false, execution reaches the last line. The multiplication needs the result of factorial(3), so this frame goes into a wait and a new call is made. The same happens for n = 3 and n = 2. In step 4, factorial(1) is called. In this frame $1 \leq 1$ is true: the base case has been reached.

The stack is now at its deepest. With main and four factorial frames there are 5 frames in total, and no multiplication has been performed yet. Each frame keeps its own value of n and waits for the result of the call above it. The upper part of Figure 12.4 shows this moment.

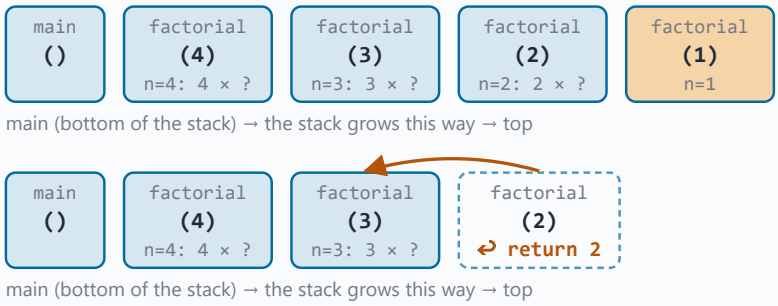


Figure 12.4: Two moments of the stack in the call factorial(4). Top: the deepest moment, with four factorial frames open, each waiting for its "n × ?" computation. Bottom: factorial(1) and factorial(2) have returned, and the frame of factorial(2) is about to return the value 2.

After step 5 the returning phase begins. factorial(1) returns 1 and its frame is removed. Control passes to the frame of factorial(2), which now has the value it was waiting for: $2 \times 1 = 2$ is computed and returned. With that value factorial(3) computes $3 \times 2 = 6$, and factorial(4) computes $4 \times 6 = 24$. Finally main prints 24.

Two conclusions follow from this trace. First, frames are closed in the reverse of the order in which they were opened. The last frame opened, factorial(1), closes first; the first one opened, factorial(4), closes last. This behavior is the "last in, first out" (LIFO) rule, and it is where the stack data structure

gets its name. Second, in a call to factorial(n) there are n factorial frames on the stack at the same time. The extra memory grows in proportion to n, that is, it is *O(n)*. For factorial(20), a version with a counter measured 20 calls and a maximum depth of 20 frames.

Common mistake

Believing that the multiplication happens while the calls are being made. Because the line `return n * factorial(n - 1);` reads from left to right, one might think the product 4 × 3 × 2 × 1 is formed in that order. But the second factor of the multiplication does not exist yet. The first multiplication to complete is *2 × 1*, then *3 × 2*, and *4 × 6* comes last. If, while tracing the call stack, you ask of each frame "what is this frame waiting for?", the order does not get confused.

12.2.3 Descent and return

A recursive computation can be thought of in two phases. In Figure 12.5, the left column is the descent phase: the calls open downward and the stack grows. The right column is the returning phase: values are carried upward and the stack shrinks.

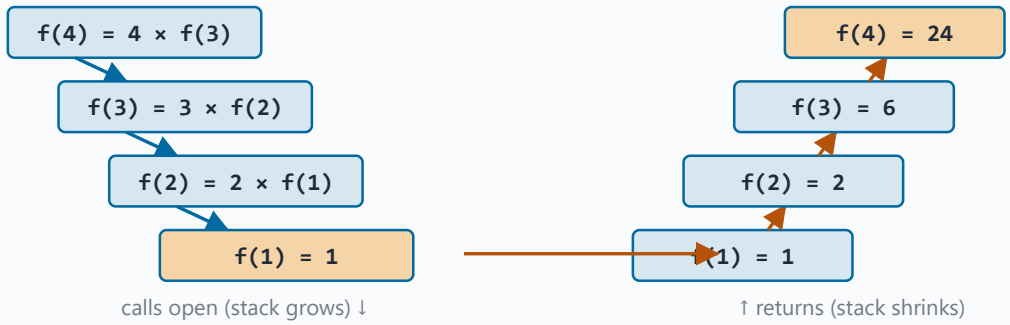


Figure 12.5: Descent and return in the call factorial(4). The calls descend until the base case is reached, and no multiplication is done in the meantime. After f(1) = 1 returns, the pending products are completed in reverse order: 2, 6, 24.

In the descent phase each call only asks a question and starts the next call. The real computation happens in the returning phase. This distinction is very useful when predicting the output of a recursive method. A print statement placed before the recursive call runs while descending. One placed after the call runs while returning, that is, in reverse order. By moving the print line, the same method can print the digits of a number in forward or reverse order.

Example 12.1

The method below prints n while n > 0, calls itself with *n − 2*, and prints n again on the way back:

```
static void g(int n) {
    if (n <= 0) return;
    System.out.print(n + " ");
    g(n - 2);
    System.out.print(n + " ");
}
```

In the call g(5), 5, 3 and 1 are printed while descending. g(−1) is the base case and returns without printing anything. In the returning phase the frames close in reverse order and 1, 3, 5 are printed. The JDK 21 output is `5 3 1 1 3 5`. The output looks like a mirror: the first half shows the descent and the second half shows the returning.

12.3 Single-call examples

In factorial, each call makes at most one new call. The calls of such methods form a single chain: f(4) → f(3) → f(2) → f(1). This shape is called **linear recursion**. All five examples in this section are linear. In each of them the same three questions are asked: what is the base case, how does the problem shrink, and how is the subresult used?

12.3.1 The sum from 1 to n

The sum of the numbers from 1 to n can be defined like factorial. sum(n) is n plus sum(n − 1). If there are no numbers, the sum is 0: *sum(0) = 0*. So *sum(3) = 3 + sum(2) = 3 + 2 + sum(1) = 3 + 2 + 1 + sum(0) = 6*.

```
static int sum(int n) {
    if (n <= 0) return 0;
    return n + sum(n - 1);
}
System.out.println(sum(3));
```

```
6
```

The base case is written as `n <= 0`. This choice makes the method safe for negative inputs as well. The call sum(−5) enters the base case at the first test and returns 0. If the condition were `n == 0`, then in the call sum(−5) n would go down through −6, −7, … and the calls would never stop.

Step	Event	Frames (including main)	Explanation
1	sum(3) called	2	3 <= 0 → false; waits for 3 + sum(2)

Step	Event	Frames (including main)	Explanation
2	sum(2) called	3	2 <= 0 → false; waits for 2 + sum(1)
3	sum(1) called	4	1 <= 0 → false; waits for 1 + sum(0)
4	sum(0) called	5	0 <= 0 → true
5	sum(0) returned	5	return 0
6	sum(1) returned	4	1 + 0 = 1
7	sum(2) returned	3	2 + 1 = 3
8	sum(3) returned	2	3 + 3 = 6
9	main prints 6	1	

Table 12.2: The trace of the call sum(3). Each addition is done after the subcall returns.

Table 12.2 closely resembles the factorial trace. In the descent phase four sum frames are opened: sum(3), sum(2), sum(1) and sum(0). At the deepest moment there are 5 frames including main. sum(0) returns 0. On the way back, *1 + 0 = 1*, *2 + 1 = 3* and *3 + 3 = 6* are computed in turn. The result is 6 = 3 + 2 + 1 + 0.

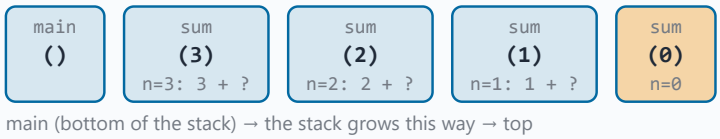


Figure 12.6: The deepest moment in the call sum(3). The sum(0) frame is at the base case; the three frames beneath it wait for the computations “3 + ?”, “2 + ?” and “1 + ?”.

The method sum(n) makes n + 1 calls and the stack gets n + 1 frames deep. Both time and extra memory are *O(n)*. The same sum can be found in a single step with the formula *n(n + 1)/2*. This example was chosen to show the mechanism of recursion; it is not the best way to compute a sum.

12.3.2 Digit sum and a helper method

The digit sum of a number can also be written recursively. The idea rests on the pattern of digits: `n % 10` gives the last digit, and `n / 10` drops the last digit. The digit sum of 472 is the last digit 2 plus the digit sum of 47. The digit sum of a one-digit number is the number itself. These sentences give both the base case and the recursive case of the method.

```
static int digitSum(int n) {
    long m = Math.abs((long) n);
    return digitSumPos(m);
}
static int digitSumPos(long n) {
    if (n < 10) return (int) n;
    return (int) (n % 10)
        + digitSumPos(n / 10);
}
System.out.println(digitSum(472));
```

The program prints 13. The code has two methods. The one that does the real work is `digitSumPos`, and it is correct only for non-negative numbers. `digitSum` is the method called from outside: it prepares the input and leaves the real work to the helper. Such a method is called a **helper method**.

Why is the preparation step needed? For a negative number, `n % 10` is negative. In Java, for example, *−472 % 10 = −2*. Also, because *−472 < 10* is true, the method would enter the base case at once and return −472. To prevent this, the absolute value is taken first. Why is the absolute value taken after converting to long? The smallest int value is −2,147,483,648, while the largest is 2,147,483,647. As a result, `Math.abs(Integer.MIN_VALUE)` does not fit in an int and again returns −2,147,483,648. `Math.abs((long) n)`, on the other hand, correctly gives 2,147,483,648. For this reason the helper takes a long parameter.

The program was also run on a few edge inputs. The second line of the output holds the results of digitSum(−472), digitSum(0) and digitSum(Integer.MIN_VALUE):

```
13
13 0 47
```

The last value is 47: *2 + 1 + 4 + 7 + 4 + 8 + 3 + 6 + 4 + 8 = 47*. If the absolute value were taken with int, the method would reach the base case with a negative number and return a wrong result.

Frame	Condition	n % 10	n / 10	Result
digitSumPos(472)	472 < 10 → false	2	47	waits
digitSumPos(47)	47 < 10 → false	7	4	waits
digitSumPos(4)	4 < 10 → true	—	—	return 4
digitSumPos(47)	return	7	4	7 + 4 = 11
digitSumPos(472)	return	2	47	2 + 11 = 13
digitSum(472)	—	—	—	return 13

Table 12.3: The trace of digitSum(472). The first three rows are the descent phase, the next three the returning phase.

Table 12.3 shows which digit each frame holds back. The frame for 472 holds back the 2, and the frame for 47 holds back the 7. The number 4 has one digit and is the base case. On the way back, *7 + 4 = 11* is computed, then *2 + 11 = 13*. The helper method returns 13, and digitSum passes this value on unchanged.



Figure 12.7: The deepest moment in the call digitSum(472). Above main there are a digitSum frame and three digitSumPos frames. The two lower helper frames hold the digits 2 and 7.

The stack depth equals the number of digits in the number. The number of digits grows logarithmically with the size of the number: an int with 10 digits opens at most 10 helper frames. So digit problems carry no risk of stack overflow.

12.3.3 Palindrome check

A palindrome is a string that reads the same backward as forward, such as “kayak”, “ana” or “abba”. Whether a string is a palindrome can be found by moving inward from both ends. If the first and last characters differ, the string is not a palindrome. If they are the same, these two characters are dropped and the remaining inner string is checked with the same question. If a single character or no character is left in the middle, the string is a palindrome.

```
static boolean isPal(String s,
                     int left, int right) {
    if (left >= right) return true;
    if (s.charAt(left) != s.charAt(right))
        return false;
    return isPal(s, left + 1, right - 1);
}
System.out.println(isPal("kayak", 0, 4));
```

The method does not make a copy of the string. Instead it describes the region to examine with two indices, `left` and `right`. On each call left increases by 1 and right decreases by 1. The measure of the problem is the length of the region, *right − left + 1*. This length drops by 2 on every call and ends up 1 or 0. The base case `left >= right` covers both of these situations.

The method has two separate exits. If the ends differ, it returns false; this also counts as a base case, because no new call is made. If the ends are equal, the result of the call in the last line is returned unchanged. No computation is done on the way back.

Frame	left >= right	Ends	Next step
isPal(s, 0, 4)	0 >= 4 → false	'k' equals 'k'	isPal(s, 1, 3)
isPal(s, 1, 3)	1 >= 3 → false	'a' equals 'a'	isPal(s, 2, 2)
isPal(s, 2, 2)	2 >= 2 → true	—	return true

Table 12.4: The trace of isPal("kayak", 0, 4).

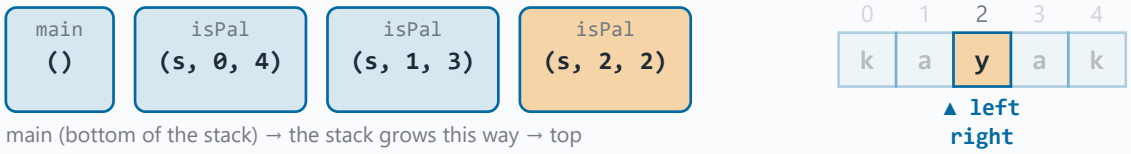


Figure 12.8: The deepest moment in the call isPal("kayak", 0, 4). Three isPal frames are open; in the topmost frame, left and right both point to the middle character 'y'.

According to Table 12.4, three calls are made for “kayak”. Characters 0 and 4 are 'k', and characters 1 and 3 are 'a'. In the third call *2 ≥ 2* is true: only 'y' is left in the middle and true is returned. This value passes through two frames and reaches main. For “kitap” a single call is enough. The first character is 'k' and the last is 'p', so the method returns false at once. The program was also run on extra inputs. The second line below holds the results for "kitap", "abba" and "a", in that order:

```
true
false true true
```

In the even-length string "abba" the inner region finally becomes empty: in the call isPal(s, 2, 1), *2 ≥ 1* is true. Writing the base case with `>=` instead of equality catches this case too. For a string of n characters at most *⌊n/2⌋ + 1* calls are made; time and stack depth are *O(n)*.

12.3.4 GCD with Euclid's algorithm

The greatest common divisor (GCD) of two numbers is the largest number that divides both without a remainder. GCD(48, 18) = 6. Euclid's algorithm finds it using division remainders. It rests on this rule: if b is nonzero, GCD(a, b) = GCD(b, a mod b). If b = 0, GCD(a, 0) = |a|. These two lines directly describe a recursive method.

```
static int gcd(int a, int b) {
    if (b == 0) return Math.abs(a);
    return gcd(b, a % b);
}
System.out.println(gcd(48, 18));
```

Why is the rule correct? Every number that divides both a and b also divides the difference $a - k \cdot b$, and a mod b is such a difference. Conversely, every number that divides both b and a mod b also divides a. So the pairs (a, b) and (b, a mod b) have the same common divisors, and two pairs with the same common divisors have the same greatest common divisor.

Frame	a	b	a % b	Next step
gcd(48, 18)	48	18	12	gcd(18, 12)
gcd(18, 12)	18	12	6	gcd(12, 6)
gcd(12, 6)	12	6	0	gcd(6, 0)
gcd(6, 0)	6	0	—	return 6

Table 12.5: The trace of gcd(48, 18). In each row a % b is computed and the next call is made with (b, a % b).

Table 12.5 shows four calls. *48 % 18 = 12*, *18 % 12 = 6* and *12 % 6 = 0*. gcd(6, 0) is the base case and returns 6. This value comes back unchanged through three frames. The last nonzero remainder, 6, is the GCD we were looking for.

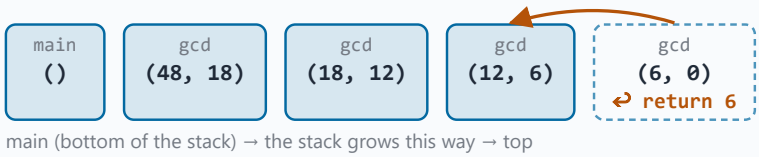


Figure 12.9: The moment the base case is reached in the call gcd(48, 18). gcd(6, 0) is about to return 6; the frames below pass this value on unchanged.

The method was also tried on other inputs. The second line below holds the results of gcd(18, 48), gcd(17, 5), gcd(−12, 8) and gcd(7, 0):

```
6
6 1 4 7
```

In the call gcd(18, 48), the first step swaps the numbers: since *18 % 48 = 18*, the next call is gcd(48, 18). That is why the version with a counter counts 5 calls on this input, against 4 for gcd(48, 18). The `Math.abs` in the base case makes the result positive for negative inputs: gcd(−12, 8) = 4.

Euclid's algorithm is very fast. The remainders at least halve every two steps, so the number of calls is proportional to the number of digits of the inputs. The worst case occurs with two consecutive Fibonacci numbers. Even for 1,134,903,170 and 701,408,733, the largest consecutive Fibonacci pair that fits in an int, the version with a counter counted only 44 calls.

12.3.5 Reversing a string

The reverse of a string can also be defined recursively. The reverse of “abc” is found by appending 'a' to the end of the reverse of “bc”. A string of length 0 or 1 is its own reverse.

```
static String reverse(String s) {
  if (s.length() <= 1) return s;
  return reverse(s.substring(1)) + s.charAt(0);
}
System.out.println(reverse("abc"));
```

`s.substring(1)` gives the remaining string without the first character, and `s.charAt(0)` is the first character. The subcall returns the reverse of the rest, and the first character is appended to that result. The addition is between a String and a char, so the result is a String.

Frame	Work	Return value
reverse("c")	length 1 ≤ 1 → base case	return "c"
reverse("bc")	reverse("c") + 'b'	"c" + 'b' = "cb"
reverse("abc")	reverse("bc") + 'a'	"cb" + 'a' = "cba"

Table 12.6: The returning phase in the call reverse("abc"). The rows follow the order in which the frames close.

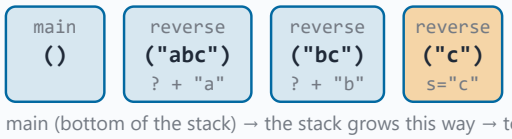


Figure 12.10: The deepest moment in the call reverse("abc"). The two lower frames wait for the concatenations “? + "a"” and “? + "b"”; the topmost one, reverse("c"), is at the base case.

The program was also run on the empty string, “kayak” and “Java”:

```
cba
[] kayak avaJ
```

The reverse of the empty string is the empty string; the square brackets are printed to make this visible. Because “kayak” is a palindrome, its reverse is the same as itself.

This method is short and readable but not efficient. Each call creates a new substring with `substring`. For a string of n characters, the total length of the substrings created is about $n(n + 1)/2$. For a string of 1000 characters, the version with a counter counted 1000 calls and substrings of 500,500 characters in total. The time is $O(n^2)$. Reversing with a loop and a `StringBuilder` takes $O(n)$. A solution being short does not mean it is cheap.

12.4 Exponentiation

12.4.1 Plain recursive power

The n-th power of a base is the product of the base with itself n times. This definition can also be written recursively: $base^n = base \times base^{n-1}$ and $base^0 = 1$. For example, $2^4 = 2 \times 2^3$.

```
static long power(long base, int n) {
  if (n == 0) return 1;
  return base * power(base, n - 1);
}
System.out.println(power(2, 4));
```

```
16
```

The structure is the same as for factorial. The base case is `n == 0`. In the recursive case n decreases by 1 and the subresult is multiplied by the base. The base stays the same on every call; only the exponent shrinks.

Frame	Condition	Return value
power(2, 0)	0 == 0 → true	return 1
power(2, 1)	1 == 0 → false	2 × 1 = 2
power(2, 2)	2 == 0 → false	2 × 2 = 4
power(2, 3)	3 == 0 → false	2 × 4 = 8
power(2, 4)	4 == 0 → false	2 × 8 = 16

Table 12.7: The returning phase of the call power(2, 4). The rows follow the order in which the frames close.

In the call power(2, 4), the calls power(2, 3), power(2, 2), power(2, 1) and power(2, 0) are opened. power(2, 0) is the base case and returns 1. Table 12.7 gives the multiplications on the way back: 2, 4, 8 and 16. In total $n + 1 = 5$ calls are made. For power(2, 10) the number is 11. Time and stack depth are $O(n)$.

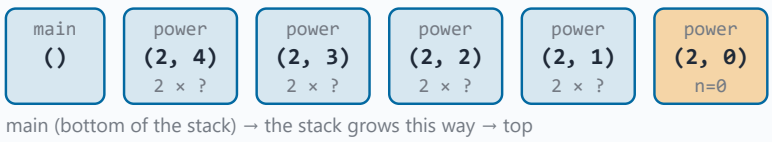


Figure 12.11: The deepest moment in the call power(2, 4). Four frames wait for the computation “2 × ?”; the

topmost one, power(2, 0), is at the base case.

What happens if the method is called with a negative exponent? In the call power(2, −1), n is −1 and *−1 == 0* is false. In the next call n is −2, then −3. n moves away from zero and the base case is never reached. The calls continue until the stack is full. This case is examined with measurements in 12.11.2.

12.4.2 Fast power

The plain method does 10 multiplications for 2^{10}. But $2^{10} = (2^5)^2$. If 2^5 is computed once and multiplied by itself, the work is halved. The same idea can be used for 2^5: since the exponent is odd, one extra base is multiplied in: $2^5 = (2^2)^2 \times 2$. Because the exponent is halved at each step, the number of calls drops sharply. This method is called **fast power**, or exponentiation by squaring.

```
static long fastPow(long base, int exp) {
  if (exp < 0) throw new IllegalArgumentException("exp < 0");
  if (exp == 0) return 1;
  long half = fastPow(base, exp / 2);
  if (exp % 2 == 0) return half * half;
  return half * half * base;
}
System.out.println(fastPow(2, 10));
```

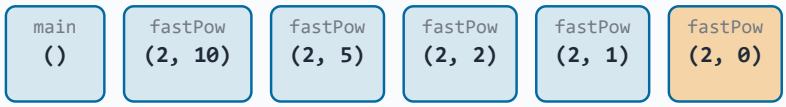
The first line rejects a negative exponent: because `exp / 2` rounds toward zero for negative numbers, −1 / 2 would be 0, and the method would return a wrong result (1) without raising any error. The method then calls itself with `exp / 2` and stores the result in the variable `half`. In Java, the quotient of two ints is an integer; *5 / 2 = 2*. If the exponent is even, the result is *half × half*. If it is odd, a 1 was lost in the division and the result is multiplied by the base once more.

The variable `half` is the most important detail of this method. If the code were written as `return fastPow(base, exp / 2) * fastPow(base, exp / 2);`, the same subproblem would be computed twice. Every call would make two calls, and the gain would vanish completely. Computing the subresult once and keeping it in a local variable is what makes the method fast.

Frame	exp / 2	half	exp	Return value
fastPow(2, 0)	—	—	—	return 1
fastPow(2, 1)	0	1	odd	1 × 1 × 2 = 2
fastPow(2, 2)	1	2	even	2 × 2 = 4
fastPow(2, 5)	2	4	odd	4 × 4 × 2 = 32
fastPow(2, 10)	5	32	even	32 × 32 = 1024

Table 12.8: The returning phase of the call fastPow(2, 10). The “exp” column shows whether the exponent is odd or even.

On the way down the exponent takes the values 10, 5, 2, 1 and 0. fastPow(2, 0) returns 1. Table 12.8 shows the way back. Exponent 1 is odd: *1 × 1 × 2 = 2*. Exponent 2 is even: *2 × 2 = 4*. Exponent 5 is odd: *4 × 4 × 2 = 32*. Exponent 10 is even: *32 × 32 = 1024*. In total 5 calls were made; the plain method would have made 11.



main (bottom of the stack) → the stack grows this way → top

Figure 12.12: The deepest moment in the call fastPow(2, 10). The exponent is halved in each frame: 10, 5, 2, 1, 0.

The number of calls depends on the number of binary digits of the exponent. For an exponent of 1 or more, the number of calls is $\lfloor \log_2 exp \rfloor + 2$. Measurements from the version with a counter agree with this formula:

Exponent	Plain power: calls	fastPow: calls	$\lfloor \log_2 exp \rfloor + 2$
10	11	5	3 + 2 = 5
16	17	6	4 + 2 = 6
62	63	7	5 + 2 = 7
1000	1001	11	9 + 2 = 11
1.000.000	1.000.001	21	19 + 2 = 21

Table 12.9: Call counts of the plain and fast power methods (measured with counting versions).

For the millionth power the plain method makes a million calls; with the default stack size this causes a stack overflow. Fast power makes only 21 calls. Time and stack depth are *O(log n)*. The program also computed fastPow(3, 16) = 43,046,721 and fastPow(2, 62) = 4,611,686,018,427,387,904 correctly:

```
1024
43046721 4611686018427387904
```

The results must fit in a long. 2^{63} exceeds the range of a long; the call fastPow(2, 63) returns −9,223,372,036,854,775,808 without raising an error. Overflow is a problem independent of the speed of the method.

12.5 Divide and conquer

Fast power reduces a problem to one of half the size. The general name for this idea is **divide and conquer**. The problem is divided into two or more smaller parts. The parts are solved with the same method, that is, by recursion. Then the solutions of the parts are combined to obtain the solution of the original problem. When the parts become small enough they are solved directly; this is the base case.

> **Definition 12.4**
>
> **Divide and conquer** is an algorithm design technique that splits a problem into smaller subproblems of the same kind, solves the subproblems recursively, and combines their solutions.

12.5.1 Recursive binary search

Binary search looks for a value in a sorted array. It examines the element in the middle of the range. If the target equals the middle element, the search ends. If the middle value is smaller, the target lies in the right half; if it is larger, the target lies in the left half. The half that was not examined is never looked at again. Binary search written with a loop can be written recursively with the same steps. Each call calls itself on half of the range.

```
static int search(int[] a, int x,
                  int lo, int hi) {
  if (lo > hi) return -1;
  int mid = lo + (hi - lo) / 2;
  if (a[mid] == x) return mid;
  if (a[mid] < x) return search(a, x, mid + 1, hi);
  return search(a, x, lo, mid - 1);
}
// a = {2, 5, 8, 12, 16, 23, 38, 56}
System.out.println(search(a, 38, 0, 7));
```

The range is described by two indices: `lo` is the lower bound and `hi` the upper bound. There are two base cases. If `lo > hi`, the range is empty and the target is not in the array; the method returns −1. If `a[mid] == x`, the value has been found and its index is returned. The other two lines halve the range. The array itself is not copied; all the calls look at the same array.

In the listing the array is given in a comment line. In the version that was run, the array was defined by the line `int[] a = {2, 5, 8, 12, 16, 23, 38, 56};`. The second line of the output holds the results of searching for 2, 56 and 20:

```
6
0 7 -1
```

Frame	lo	hi	mid	a[mid]	Decision
search(a, 38, 0, 7)	0	7	3	12	12 < 38: right half, search(a, 38, 4, 7)
search(a, 38, 4, 7)	4	7	5	23	23 < 38: right half, search(a, 38, 6, 7)
search(a, 38, 6, 7)	6	7	6	38	38 == 38: return 6

Table 12.10: The trace of search(a, 38, 0, 7).

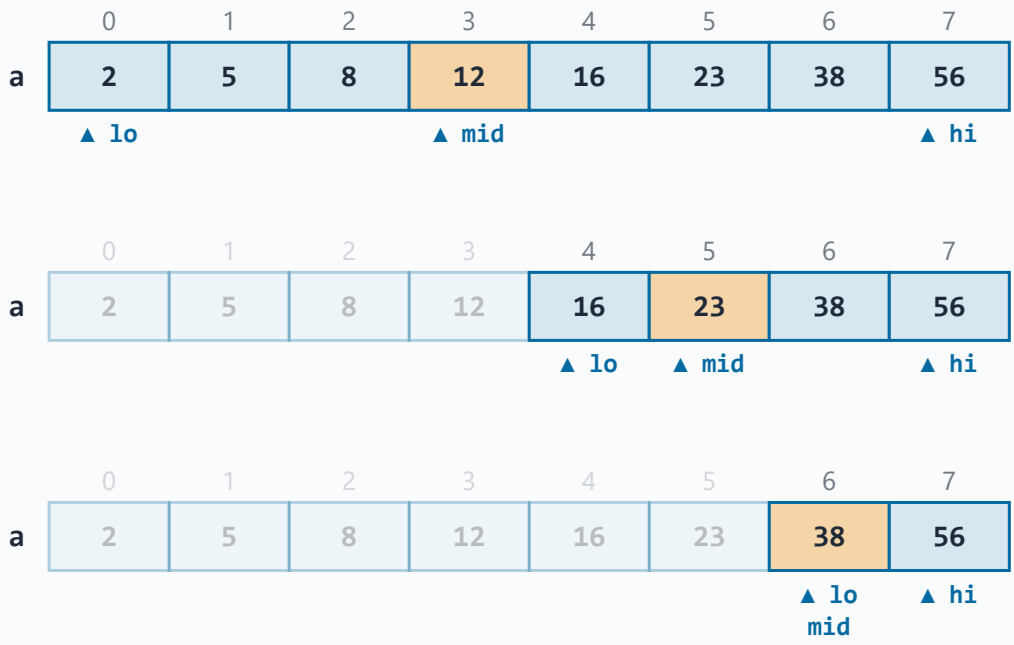


Figure 12.13: Searching for 38. Top: the first call, lo = 0, hi = 7, mid = 3. Middle: the second call, lo = 4, hi = 7, mid = 5. Bottom: the third call, lo = 6, hi = 7, mid = 6 and a[6] = 38.

According to Table 12.10, 38 is found in three calls. In the first call *mid = 0 + (7 − 0) / 2 = 3* and a[3] = 12. Since 12 < 38, the search continues in the right half. In the second call mid = 5 and a[5] = 23; the search again moves right. In the third call mid = 6 and a[6] = 38: the value has been found. The 6 passes through two frames and reaches main.

When a value that is not in the array is searched for, the range eventually becomes empty. Table 12.11 shows the search for 20. In the fourth call lo = 5 and hi = 4; *5 > 4* is true and −1 is returned.

Frame	lo	hi	mid	a[mid]	Decision
search(a, 20, 0, 7)	0	7	3	12	12 < 20: right half, search(a, 20, 4, 7)
search(a, 20, 4, 7)	4	7	5	23	23 > 20: left half, search(a, 20, 4, 4)
search(a, 20, 4, 4)	4	4	4	16	16 < 20: right half, search(a, 20, 5, 4)

Frame	lo	hi	mid	a[mid]	Decision
search(a, 20, 5, 4)	5	4	—	—	5 > 4: return −1

Table 12.11: The trace of search(a, 20, 0, 7). 20 is not in the array.

Because the range is halved on every call, an array of n elements needs at most about $log_2\ n + 1$ calls. For an array of 8 elements the worst case is 4 calls: the searches for 56 and 20 took 4 calls. The time is *O(log n)*. The extra stack memory of the recursive version is also *O(log n)*; the loop version does the same job with *O(1)* extra memory.

Common mistake

Forgetting to write `return` in front of a recursive call. In the line `if (a[mid] < x) search(a, x, mid + 1, hi);` the subcall finds the value but the value is not used. The method continues to the lines below and returns a wrong result. In a recursive method that returns a value, the result of the subcall must either be returned or used in a computation.

12.5.2 Merge sort

Merge sort is a typical example of divide and conquer. The array is split in the middle. Each half is sorted recursively. Then the two sorted halves are merged into a single sorted array. An array with one element is already sorted; this is the base case.

```
static void mergeSort(int[] a, int lo, int hi) {
    if (lo >= hi) return;
    int mid = (lo + hi) / 2;
    mergeSort(a, lo, mid);
    mergeSort(a, mid + 1, hi);
    int[] t = new int[hi - lo + 1];
    int i = lo, j = mid + 1, k = 0;
    while (i <= mid && j <= hi)
        t[k++] = a[i] <= a[j] ? a[i++] : a[j++];
    while (i <= mid) t[k++] = a[i++];
    while (j <= hi) t[k++] = a[j++];
    System.arraycopy(t, 0, a, lo, t.length);
}
```

The method has three parts:

1. **Base case:** if `lo >= hi`, the range has at most one element and the method returns at once.
2. **Split:** `mid` is computed; `mergeSort(a, lo, mid)` sorts the left half and `mergeSort(a, mid + 1, hi)` sorts the right half.

3. **Merge:** a temporary array named `t` is created. `i` advances through the left half and `j` through the right half. At each step the smaller of the two front elements is written to t. When one half runs out, the rest of the other is appended as it is. Finally `System.arraycopy` copies t back into the corresponding region of the array.

The “less than or equal” in the comparison `a[i] <= a[j]` is a deliberate choice. For equal elements, the one on the left is written first, so the original order of equal values is preserved. This property is called stability, and merge sort is a stable sorting method.

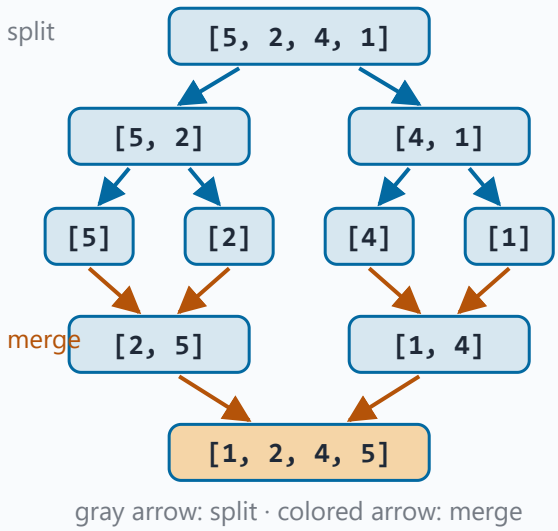


Figure 12.14: Merge sort on the array {5, 2, 4, 1}. Gray arrows show splitting and colored arrows show merging. [5, 2] → [2, 5] and [4, 1] → [1, 4]; finally [1, 2, 4, 5] is obtained.

Call no.	Frame	Work done	Array
1	mergeSort(a, 0, 3)	mid = 1: [0..1] and [2..3] to be sorted	[5, 2, 4, 1]
2	mergeSort(a, 0, 1)	mid = 0: [0..0] and [1..1] to be sorted	[5, 2, 4, 1]
3	mergeSort(a, 0, 0)	single element: returns at once	[5, 2, 4, 1]
4	mergeSort(a, 1, 1)	single element: returns at once	[5, 2, 4, 1]
2	mergeSort(a, 0, 1)	merge: [2, 5]	[2, 5, 4, 1]
5	mergeSort(a, 2, 3)	mid = 2: [2..2] and [3..3] to be sorted	[2, 5, 4, 1]
6	mergeSort(a, 2, 2)	single element: returns at once	[2, 5, 4, 1]
7	mergeSort(a, 3, 3)	single element: returns at once	[2, 5, 4, 1]
5	mergeSort(a, 2, 3)	merge: [1, 4]	[2, 5, 1, 4]
1	mergeSort(a, 0, 3)	merge: [1, 2, 4, 5]	[1, 2, 4, 5]

Table 12.12: The order of the calls in mergeSort(a, 0, 3), with a = {5, 2, 4, 1}. The same call number marks the two moments at which a frame first splits and later merges.

Table 12.12 gives the true order of the calls. Call 1 splits the array into [0..1] and [2..3]. The right half is not started until the left half is completely finished. Call 2 splits [5, 2] into two single elements, both return at once, and then the merge produces [2, 5]. The array is now [2, 5, 4, 1]. The right half turns into [1, 4] in the same way. Finally, call 1 merges [2, 5] with [1, 4]. The comparisons are between 2 and 1, 2 and 4, and 5 and 4; when the right half runs out, 5 is appended. The result is [1, 2, 4, 5].

The program also sorted an array of 7 elements:

```
[1, 2, 4, 5]
[3, 9, 10, 27, 38, 43, 82]
```

The cost is computed as follows. During splitting, the size of the array halves at each level. An array of n elements is broken into single elements in about $log_2\ n$ levels. Merging all the pieces at each level requires writing n elements in total. The total work is the number of levels times the work per level: $O(n \log n)$. Because a temporary array is used for merging, the extra memory is $O(n)$. The stack depth, however, is only $O(\log n)$. For a reverse-sorted array of 1024 elements, the version with a counter counted 2047 calls, a maximum depth of 11 frames and 5120 comparisons. $5120 = 512 \times 10$, that is, $(n/2) \cdot log_2\ n$.

For comparison: methods with two nested loops, such as selection sort or bubble sort, make about $n^2/2$ comparisons. For 1024 elements that means roughly 520 thousand comparisons. Merge sort sorts the same array with a hundred times fewer comparisons.

12.6 Recursion with two calls

In most of the examples so far, each call made at most one new call. Merge sort makes two calls: one for the left half and one for the right half. In methods where every call makes two calls, the calls form a tree rather than a chain. This tree is called a **recursion tree**. The root of the tree is the first call, and the children of each node are the subcalls that call makes. The Towers of Hanoi and subset generation are two classic examples of this structure.

12.6.1 Towers of Hanoi

The Towers of Hanoi is a puzzle. There are three pegs: A, B and C. Peg A holds n disks arranged from largest to smallest. The goal is to move all the disks to peg C. There are two rules: only one disk may be moved at a time, and it must be the top disk of a peg. A larger disk may never be placed on a smaller one.

The puzzle looks hard at first, but its recursive solution takes three lines. To move n disks from A to C, first the top n − 1 disks are moved to B, with C serving as the spare peg. Then the largest disk, left alone on A, is moved to C. Finally the n − 1 disks on B are moved to C, with A as the spare. Moving n − 1 disks is a smaller instance of the same problem. If there are no disks, there is nothing to do.

```
static void hanoi(int n, char from,
                  char to, char via) {
  if (n == 0) return;
  hanoi(n - 1, from, via, to);
  System.out.print(from + "->" + to + " ");
  hanoi(n - 1, via, to, from);
}
hanoi(3, 'A', 'C', 'B');
```

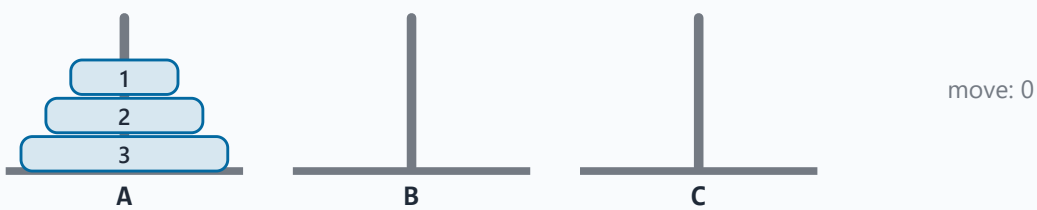
```
A->C A->B C->B A->C B->A B->C A->C
```

The method has four parameters: the number of disks, the source peg, the target peg and the helper (via) peg. In the first recursive call, to and via swap places: n − 1 disks go to the via peg. In the second call, from and via swap places: n − 1 disks go from the via peg to the target. The print statement sits between the two calls, so a move is printed only after the left subtree has finished completely.

Move	Disk	Transfer	Frame that makes the move
1	1	A → C	hanoi(1, A, C, B)
2	2	A → B	hanoi(2, A, B, C)
3	1	C → B	hanoi(1, C, B, A)
4	3	A → C	hanoi(3, A, C, B)
5	1	B → A	hanoi(1, B, A, C)
6	2	B → C	hanoi(2, B, C, A)
7	1	A → C	hanoi(1, A, C, B)

Table 12.13: The moves of hanoi(3, 'A', 'C', 'B'). The last column is the frame that prints the move.

Table 12.13 shows the seven moves and which call makes each one. Disk 1 is the smallest and disk 3 the largest. The first three moves carry the top two disks to B. Move 4 takes the largest disk to C; this move is made by the outermost call, hanoi(3, A, C, B). The last three moves carry the two disks on B to C. Disk 1 moves on every other move: moves 1, 3, 5 and 7 are its. Figure 12.15 shows four moments.



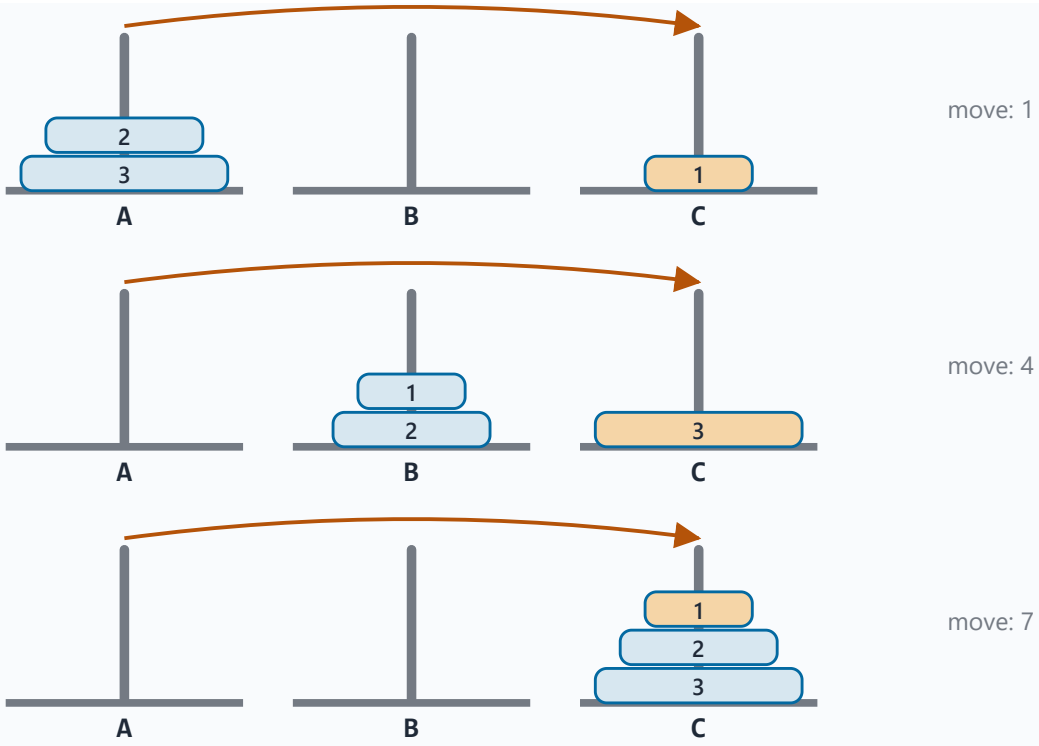


Figure 12.15: Four moments of the three-disk Towers of Hanoi. From top to bottom: the start, after move 1 (disk 1 on C), after move 4 (disk 3 on C, disks 1 and 2 on B), and after move 7 (all disks on C).

The number of moves can also be found recursively. Let h(n) be the number of moves needed to move n disks. The solution moves n − 1 disks twice and moves the largest disk once: $h(n) = 2 \cdot h(n-1) + 1$, $h(0) = 0$. From this, h(1) = 1, h(2) = 3, h(3) = 7, h(4) = 15 and h(5) = 31. The general formula is $h(n) = 2^n - 1$. The formula is verified by induction: $2 \cdot (2^{n-1} - 1) + 1 = 2^n - 1$.

Number of disks n	Move	$2^n - 1$	Calls	Maximum depth
1	1	1	3	2
2	3	3	7	3
3	7	7	15	4
4	15	15	31	5
5	31	31	63	6
10	1023	1023	2047	11
20	1.048.575	1.048.575	2.097.151	21

Table 12.14: Move and call counts of the Hanoi solution (measured with the counting version; calls to hanoi(0) included).

Table 12.14 shows two different kinds of growth. The number of moves and calls is exponential: it roughly doubles with each added disk. The stack depth is linear: n + 1. This solution of Hanoi is optimal. It is impossible to move n disks in fewer than $2^n - 1$ moves, because for the largest disk to

move once, the n − 1 disks on top of it must be moved away completely twice. The exponential growth is a property of the problem itself, not of the algorithm. For 64 disks the number of moves needed is $2^{64} - 1 \approx 1.8 \times 10^{19}$.

12.6.2 Generating subsets and backtracking

The set {1, 2} has four subsets: {1, 2}, {1}, {2} and the empty set. A set of n elements has 2^n subsets. The reason is simple: for each element there are two choices, either it is included in the subset or it is not. For n elements the choices multiply: $2 \times 2 \times \ldots \times 2 = 2^n$.

This idea of "two choices for each element" becomes a recursive method. The method decides about the i-th element. First it adds the element to the list and calls itself for the remaining elements. Then it removes the element from the list and calls itself again for the remaining elements. If a decision has been made for all the elements, the list is a subset and it is printed.

```
static void subsets(int[] a, int i,
                    List<Integer> chosen) {
    if (i == a.length) {
        System.out.println(chosen);
        return;
    }
    chosen.add(a[i]);
    subsets(a, i + 1, chosen);
    chosen.remove(chosen.size() - 1);
    subsets(a, i + 1, chosen);
}
subsets(new int[]{1, 2}, 0, new ArrayList<>());
```

`chosen` is a list that holds the elements selected so far (`List<Integer>`). All the frames share the same list object; each call receives the list itself, not a copy. `chosen.add(a[i])` appends the element at the end. `chosen.remove(chosen.size() - 1)` removes the last element and returns the list to its state before the addition. Without this removal step, the second call would find the element added by the first call still in the list.

Definition 12.5

Backtracking is a search technique that makes choices step by step, explores the consequences of a choice recursively, and then undoes that choice and tries the next option.

Step	Frame	Work	chosen	Output
1	subsets(i=0)	1 included: chosen.add(1)	[1]	
2	subsets(i=1)	2 included: chosen.add(2)	[1, 2]	
3	subsets(i=2)	base case: print	[1, 2]	[1, 2]

Step	Frame	Work	chosen	Output
4	subsets(i=1)	2 excluded: chosen.remove(...)	[1]	[1, 2]
5	subsets(i=2)	base case: print	[1]	[1, 2] [1]
6	subsets(i=0)	1 excluded: chosen.remove(...)	[]	[1, 2] [1]
7	subsets(i=1)	2 included: chosen.add(2)	[2]	[1, 2] [1]
8	subsets(i=2)	base case: print	[2]	[1, 2] [1] [2]
9	subsets(i=1)	2 excluded: chosen.remove(...)	[]	[1, 2] [1] [2]
10	subsets(i=2)	base case: print	[]	[1, 2] [1] [2] []

Table 12.15: The trace of subsets(new int[]{1, 2}, 0, ...). The "chosen" column is the state of the shared list after that step.

Table 12.15 shows in ten steps how the four subsets are generated. In steps 1 and 2 both elements are added and the list becomes [1, 2]. In step 3, i = 2, so the end of the array has been reached: [1, 2] is printed. In step 4 the method backs up and 2 is removed. The list becomes [1], and the "2 excluded" branch prints [1]. In step 6 control returns to the frame with i = 0 and 1 is removed as well. After that, the "1 excluded" branches produce the subsets [2] and [].

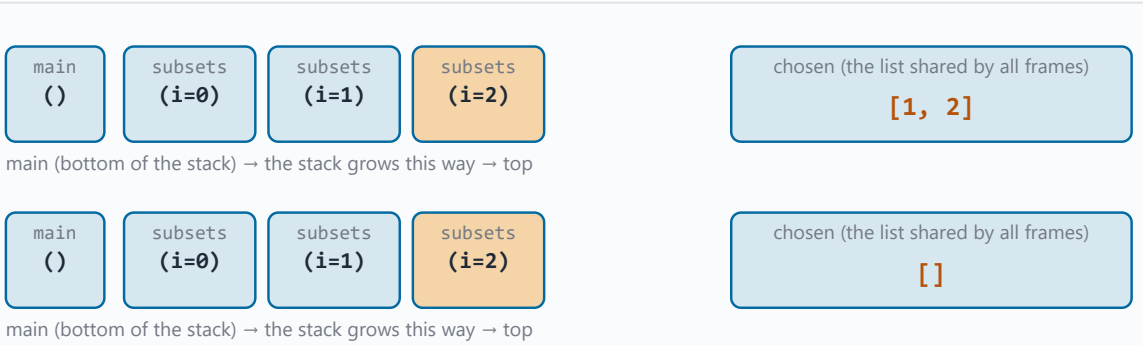


Figure 12.16: Two leaves of subset generation. Top: the first leaf, with three subsets frames open and the shared list [1, 2]. Bottom: the last leaf, where the list is empty and the empty set is printed.

The output of the program contains first the subsets of the two-element set and, after the "--" line, the subsets of {1, 2, 3}:

```
[1, 2]
[1]
[2]
[]
--
[1, 2, 3]
[1, 2]
[1, 3]
[1]
[2, 3]
[2]
[3]
[]
```

The output order is the left-to-right traversal order of the recursion tree. At each level the "included" branch runs first, then the "excluded" branch. This is why the first line is the set containing all the elements and the last line is the empty set.

Each level of the recursion tree corresponds to one element. For an array of n elements the tree has 2^n leaves and $2^{n+1} - 1$ nodes in total. The version with a counter counted 7 calls for n = 2, 15 for n = 3 and 2047 for n = 10. The stack depth is only n + 1. The time is $O(2^n)$, but this is the price that any method that wants to produce all 2^n subsets must pay.

12.7 The call stack and stack overflow

12.7.1 What does a stack frame hold?

All the earlier examples used the call stack. This section explains exactly what is stored on the stack and why the stack is limited. When a method calls another method, the work of the calling method is left half finished and will continue later from where it stopped. Three pieces of information must be saved for this:

1. **Parameters:** the values passed to the call. In the factorial(3) frame, n = 3.
2. **Local variables:** variables defined inside the method. `half` in fastPow, or `mid`, `i` and `j` in mergeSort, are such variables.
3. **Return point:** the instruction in the calling method at which to continue when the called method finishes. When factorial(3) finishes, execution returns to the multiplication in the frame of factorial(4).

The Java Virtual Machine (JVM) keeps this information in a separate stack frame for each call. The frames sit on top of one another in the stack memory reserved for the thread. When a new call is made, a frame is added on top. When a method ends with `return`, the top frame is removed and

control passes to the frame below. Frames therefore close in reverse order: the last one opened closes first.

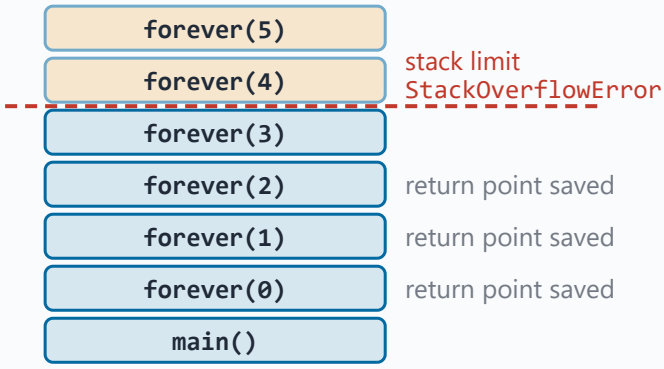


Figure 12.17: In recursion with no base case, the frames exceed the stack limit. Each frame stores its return point. When there is no room left for a frame that would go over the limit, a StackOverflowError occurs.

Stack memory is limited. In Java, the stack of each thread has a fixed size set when the program starts. On 64-bit systems this default size is usually 1 MB. If the frames pile up past the limit, the JVM cannot find room for a new frame and throws an error. This situation is called **stack overflow**.

Definition 12.6

Stack overflow occurs when the frames on the call stack fill the memory set aside for the stack. In Java, stack overflow is reported with a `java.lang.StackOverflowError`. Recursion that never reaches the base case is called **infinite recursion**; infinite recursion always ends in a stack overflow.

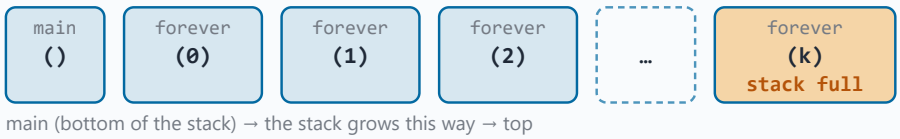
Stack overflow has two causes. The first is infinite recursion: there is no base case, it is written wrongly, or the problem does not shrink. The second is recursion that is written correctly but goes very deep. The base case will be reached, but the frames accumulated by then do not fit on the stack. Linear recursion opens as many frames as the size of the input, so even a correct method can overflow on large inputs.

12.7.2 StackOverflowError

The shortest example in the presentation is a method that contains no condition at all:

```
static void forever(int n) {
    forever(n + 1);
}
forever(0);
```

The call forever(0) immediately calls forever(1), which calls forever(2), and so on. Since no call ever returns, no frame is ever removed. Figure 12.18 shows this accumulation. When the stack is full, the JVM throws a StackOverflowError. Because the error is not caught, the program ends and the error text is written to the standard error stream (stderr). The first lines of the error text of the program run on JDK 21 are:



main (bottom of the stack) → the stack grows this way → top

Figure 12.18: The stack fills up in the call forever(0). Frames accumulate as forever(0), forever(1), forever(2), …; at the k-th call the stack is full.

```
Exception in thread "main" java.lang.StackOverflowError
    at S19.forever(S19.java:5)
    at S19.forever(S19.java:5)
    at S19.forever(S19.java:5)
    ...
```

Each “at” line shows one frame on the stack: it gives the class and method and the line of the source file. The line number is 5 because in the program the method forever makes its call on line 5 of the class. These lines repeat thousands of times. By default the JVM writes at most 1024 frames into the error text; the error text of the program that was run also consisted of 1 header line and 1024 “at” lines. When the same line repeats again and again in such an error text, it is a clear sign of infinite or very deep recursion.

The number of calls after which the overflow occurs is not fixed. The size of a frame depends on the number of parameters and local variables of the method. Whether the JVM runs the code by interpreting it or by compiling it to machine code also changes the frame size. The program below counts the calls made up to the moment of overflow. Because the error is caught with `catch`, the program does not end and can print the count:

```
// Counts how many calls it takes to get a StackOverflowError at the default stack size
(varies with the JVM and conditions).
public class Depth {
    static int count;

    static void forever(int n) {
        count++;
        forever(n + 1);
    }

    static long power(long base, int n) {
        count++;
        if (n == 0) return 1;
        return base * power(base, n - 1);
    }

    static int sum(int n) {
        if (n <= 0) return 0;
        return n + sum(n - 1);
    }

    public static void main(String[] args) {
        for (int k = 0; k < 3; k++) {
            count = 0;
            try {
                forever(0);
            } catch (StackOverflowError e) {
                System.out.println("forever: " + count + " calls until " +
e.getClass().getName());
            }
        }
        count = 0;
        try {
            power(2, -1);
        } catch (StackOverflowError e) {
            System.out.println("power(2, -1): " + count + " calls until " +
e.getClass().getSimpleName());
        }
        for (int n : new int[]{1000, 5000, 10000, 100000, 1000000}) {
            try {
                System.out.println("sum(" + n + ") = " + sum(n));
            } catch (StackOverflowError e) {
                System.out.println("sum(" + n + "): StackOverflowError");
            }
        }
    }
}
```

The output of one run of the program is:

```
forever: 23539 calls until java.lang.StackOverflowError
forever: 62252 calls until java.lang.StackOverflowError
forever: 62252 calls until java.lang.StackOverflowError
power(2, -1): 8079 calls until StackOverflowError
sum(1000) = 500500
sum(5000) = 12502500
sum(10000) = 50005000
sum(100000): StackOverflowError
sum(1000000): StackOverflowError
```

When the same method was tried three times in the same program, it overflowed after about 23 thousand calls on the first attempt and after about 62 thousand on the later ones. When the program was run several more times, the count of the first attempt varied between 22 thousand and 27 thousand. The numbers may come out slightly different on each run, but the order of magnitude is clear: a few tens of thousands of calls at the default stack size. sum(10,000) works, while sum(100,000) overflows. The stack size can be increased with an option such as `java -Xss4m Program`. This only pushes the limit further away; it does not solve the problem of deep recursion.

Common mistake

Catching StackOverflowError with `catch` and trying to carry on with the program. This error is a symptom of a programming mistake or of a badly chosen algorithm. The program above catches the error only to take measurements. In a real program the fix is to correct the base case, to check the input, or to turn deep recursion into a loop.

12.8 Recursion or a loop?

12.8.1 Criteria for choosing

Every method written with recursion can also be written with a loop, and every method written with a loop can also be written with recursion. The choice between the two depends not on which one is more “powerful” but on the structure of the problem and on cost.

Recursion is the natural choice for problems that resemble themselves. Tree-shaped data, directory and folder structures, divide-and-conquer algorithms and backtracking are of this kind. For these problems the recursive code is very close to the mathematical definition and easy to read. It is possible to write Hanoi and subset generation with a loop, but then you have to manage by hand a stack data structure that does the job of the call stack.

For simple linear repetition, on the other hand, the loop is better. Factorial, the sum from 1 to n and the digit sum are written just as clearly with a loop. A loop runs in a single frame and does not open a new frame for each iteration. Its extra stack memory is *O(1)*, and no matter how large the input, the

stack does not overflow. Making a call also has a small time cost: opening and closing a frame is more expensive than one iteration of a loop.

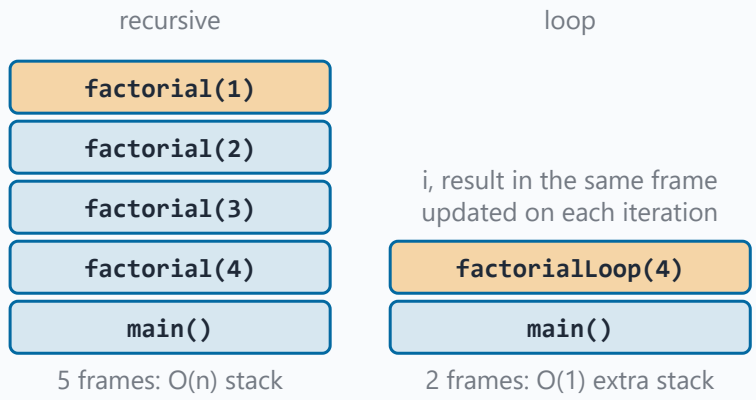


Figure 12.19: Stack use for factorial(4). The recursive version uses 5 frames, counting main. The loop version uses 2 frames; i and result are updated in the same frame on each iteration.

The choice should not be made by looking only at how short the code is. Three criteria are weighed together: correctness, readability and resource use. A recursive solution with a clear base case and a clear shrinking step can reduce the chance of mistakes. But if the size of the input can exceed tens of thousands, linear recursion puts the stack at risk. A recursion that solves the same subproblem many times puts the time at risk; this is seen in detail in the Fibonacci example.

Property	Recursive	Loop
Time complexity	O(n)	O(n)
Extra stack memory	O(n): n frames	O(1): a single frame
Flow	Close to the mathematical definition	Explicit counter and accumulator
Risk of depth	Yes: the stack can overflow for large n	None

Table 12.16: Recursive and loop factorial compared.

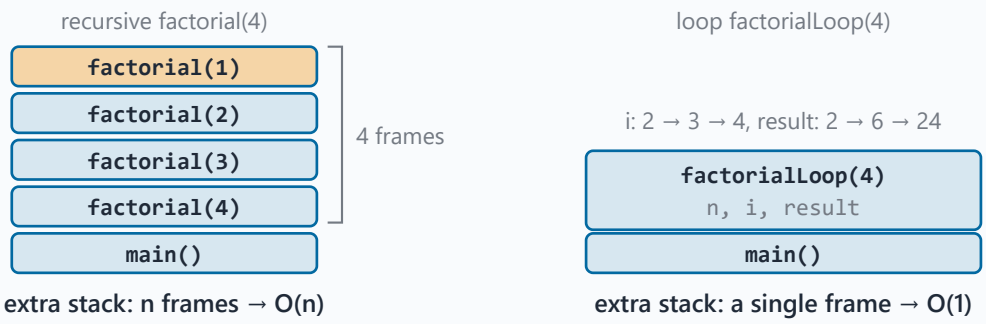


Figure 12.20: The stack for n = 4. On the left, the 4 factorial frames of the recursive version (extra stack O(n)); on the right, the single frame of the loop version (extra stack O(1)); in the loop, i goes from 2 to 4 and result takes the values 2, 6, 24.

12.8.2 Factorial with a loop

The loop version of factorial works with an accumulator. `result` starts at 1 and is multiplied by the counter i on every iteration:

```
static long factorialLoop(int n) {
    long result = 1;
    for (int i = 2; i <= n; i++)
        result *= i;
    return result;
}
System.out.println(factorialLoop(4));
```

```
24
```

Iteration	i	i <= n	result (after the body)
1	2	`true`	2
2	3	`true`	6
3	4	`true`	24
—	5	`false`	24 (loop ends)

Table 12.17: The trace of factorialLoop(4). All iterations run in the same frame.

Table 12.17 shows three iterations. At i = 2 result is 2, at i = 3 it is 6, and at i = 4 it is 24. At i = 5, $5 \le 4$ is false and the loop ends. Starting the loop at i = 2 is deliberate: multiplying by 1 does not change the result. For n = 0 and n = 1 the loop never runs and 1 is returned, which agrees with the definition 0! = 1! = 1.

Compared with the recursive version, the computation is the same: $2 \times 3 \times 4 = 24$. The difference lies in the order of the multiplications and in stack use. In the loop version the multiplications are done going forward, from small to large. In the recursive version the calls open from large to small and the multiplications are done on the way back. In the loop version the stack never exceeds 2 frames. A `while` version of the same method does the same job at the same cost; its results for n = 0 to 5 are 1, 1, 2, 6, 24 and 120.

12.9 Fibonacci and memoization

12.9.1 Direct Fibonacci

The Fibonacci sequence starts with 0 and 1. Each following term is the sum of the two terms before it: 0, 1, 1, 2, 3, 5, 8, 13, 21, … The definition is recursive: *fib(0) = 0*, *fib(1) = 1* and *fib(n) = fib(n − 1) + fib(n − 2)*. The method translated directly into Java is:

```
static long fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
System.out.println(fib(4));
```

```
3
```

The method is a word-for-word rendering of the definition and gives the correct result. But it differs from factorial in one important way: in the recursive case two calls are made. First fib(n − 1) is computed completely, then fib(n − 2). The calls form a tree, not a chain.

Call no.	Frame	Depth	Explanation
1	fib(4)	1	waits for fib(3) + fib(2)
2	fib(3)	2	waits for fib(2) + fib(1)
3	fib(2)	3	waits for fib(1) + fib(0)
4	fib(1)	4	base case: return 1
5	fib(0)	4	base case: return 0
	fib(2) returns	3	1 + 0 = 1
6	fib(1)	3	base case: return 1
	fib(3) returns	2	1 + 1 = 2
7	fib(2)	2	waits for fib(1) + fib(0)
8	fib(1)	3	base case: return 1
9	fib(0)	3	base case: return 0
	fib(2) returns	2	1 + 0 = 1
	fib(4) returns	1	2 + 1 = 3

Table 12.18: The order of the calls in fib(4). The “Depth” column is the number of frames excluding main; rows without a number are the moments at which a frame adds its two subresults and returns.

Table 12.18 gives the order of the nine calls. fib(4) calls fib(3), fib(3) calls fib(2), and fib(2) calls fib(1). fib(1) and fib(0) are base cases. fib(2) = 1 + 0 = 1 is computed. For its right term, fib(3) calls fib(1) and finds 1 + 1 = 2. Then fib(4) calls fib(2) for its right term. This call, number 7, redoes exactly the work done by call number 3. fib(2) is computed twice, fib(1) three times and fib(0) twice. Although the result is 3, 9 calls were made. The stack depth is at most 4, however: at any moment only the frames along one path from the root to a leaf are open.

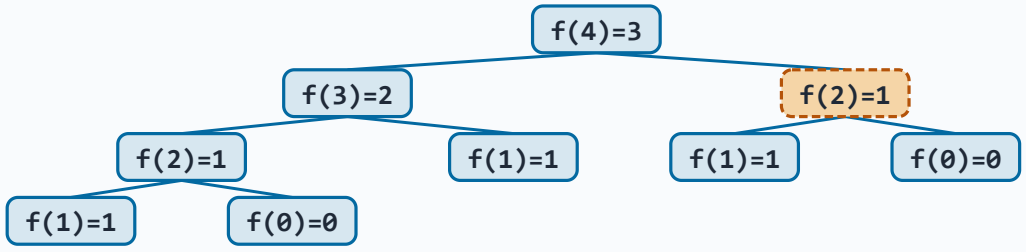


Figure 12.21: The completed recursion tree of the call fib(4). The value each node returned is written inside it. The f(2) in the right subtree repeats a subproblem that was already computed in the left subtree.

As n grows, the repetition increases quickly. Figure 12.22 shows the tree for fib(5). 15 calls are made: f(3) is computed 2 times, f(2) 3 times, f(1) 5 times and f(0) 3 times. The f(3) subtree in the right half of the tree is already contained in the f(4) subtree in the left half. The subtrees of fib(n − 1) and fib(n − 2) overlap to a large extent.

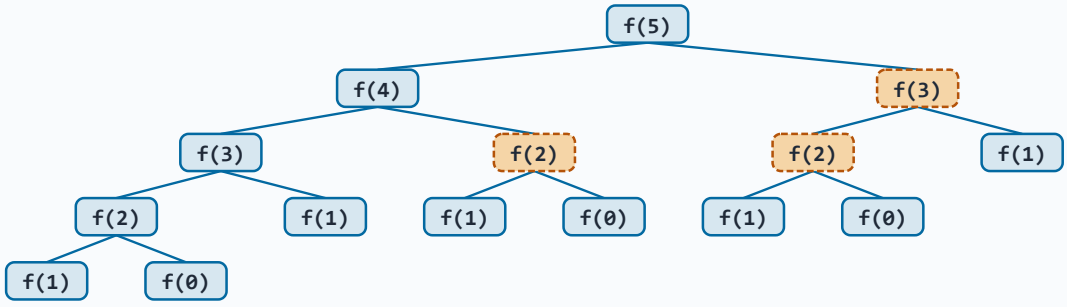


Figure 12.22: The recursion tree of fib(5): 15 calls. The dashed orange nodes repeat a subproblem that was computed earlier.

12.9.2 How the call count grows

The call count is found most reliably by counting. In the program below, a counter has been added to the method. Every call increases the counter by 1:

```
public class FibCounter {
    static long calls = 0;

    static long fib(int n) {
        calls++;
        if (n <= 1) return n;
        return fib(n - 1) + fib(n - 2);
    }

    public static void main(String[] args) {
        for (int n : new int[]{5, 10, 20, 30}) {
            calls = 0;
            long result = fib(n);
            System.out.println("fib(" + n + ") = " + result + ", calls = " + calls);
        }
    }
}
```

```
fib(5) = 5, calls = 15
fib(10) = 55, calls = 177
fib(20) = 6765, calls = 21891
fib(30) = 832040, calls = 2692537
```

The numbers can also be found with a formula. Let c(n) be the number of calls made for fib(n). The first call counts as 1; the calls of the two subtrees are added to it: $c(n) = 1 + c(n-1) + c(n-2)$, $c(0) = c(1) = 1$. From this, c(2) = 3, c(3) = 5, c(4) = 9, c(5) = 15 and c(6) = 25. The solution of this recurrence is $c(n) = 2 \cdot fib(n+1) - 1$. For example, since fib(11) = 89, c(10) = 177.

n	fib(n)	Recursive: number of calls	Loop: number of iterations
5	5	15	5
10	55	177	10
20	6765	21.891	20
30	832.040	2.692.537	30
40	102.334.155	331.160.281	40

Table 12.19: The amount of work of recursive and loop Fibonacci. The numbers were obtained by incrementing a counter on every call and every iteration; they are not timings.

Table 12.19 puts the two kinds of growth side by side. The loop version runs n iterations; its work is proportional to n. The call count of the recursive version, however, is multiplied by about 123 each time n grows by 10. The reason is the Fibonacci numbers themselves. The ratio of two consecutive Fibonacci numbers approaches about 1.618. This number is called the golden ratio and is written φ. $\varphi^{10} \approx 123$. Since the call count is proportional to fib(n + 1), it also grows by a factor of about 1.618 at

each step. The time is $O(\varphi^n)$. The $O(2^n)$ notation used in the slides is a rough upper bound on this: every call makes at most two calls and the depth of the tree is n. Whichever is used, the growth is exponential.

The practical meaning of exponential growth is this: for fib(40), 331 million calls are made. For fib(50) the number exceeds 40 billion. The loop version finds fib(50) in 50 iterations. Stack depth is not a problem in either version: the recursive version opens at most n frames. The problem is time.

Common mistake

Believing that every recursive method that matches the definition exactly is efficient. The direct Fibonacci method is correct but solves the same subproblems again and again. If several nodes with the same argument appear in the recursion tree, the method is doing unnecessary work. This repetition is removed by saving the results, or by computing from small to large with a loop.

12.9.3 Memoization

The cure for repetition is simple: when the result of a subproblem is computed for the first time, it is written down somewhere. When the same subproblem is requested again, no computation is done; the recorded value is returned. This technique is called **memoization**.

Definition 12.7

Memoization is the technique in which a recursive method stores the subproblem results it has computed in an array or table and, when the same input comes again, returns the recorded value without expanding the recursion tree again.

For Fibonacci, the place to record results is a `long` array. memo[k] holds the result of fib(k). In Java, all the cells of a new numeric array start at 0. Since fib(k) ≥ 1 for k ≥ 1, the value 0 can be used to mean “not computed yet”.

```
static long fib(int n, long[] memo) {
    if (n <= 1) return n;
    if (memo[n] != 0) return memo[n];
    memo[n] = fib(n - 1, memo)
              + fib(n - 2, memo);
    return memo[n];
}
long[] memo = new long[6];
System.out.println(fib(5, memo));
```

```
5
```

The method differs from the plain version in only three places:

1. It takes the array as a parameter. All the calls share the same array object; the values written by one call are seen by the others.
2. After the base case it looks at the record: if `memo[n] != 0`, the result is already known and is returned at once.
3. It saves the result before returning it: `memo[n] = fib(n - 1, memo) + fib(n - 2, memo);`.

The call count of the memoized version was also measured with a counter:

```
public class MemoCounter {
    static long calls = 0;

    static long fib(int n, long[] memo) {
        calls++;
        if (n <= 1) return n;
        if (memo[n] != 0) return memo[n];
        memo[n] = fib(n - 1, memo)
                  + fib(n - 2, memo);
        return memo[n];
    }

    public static void main(String[] args) {
        for (int n : new int[]{5, 10, 20, 30, 50}) {
            calls = 0;
            long result = fib(n, new long[n + 1]);
            System.out.println("fib(" + n + ") = " + result + ", calls = " + calls);
        }
    }
}
```

```
fib(5) = 5, calls = 9
fib(10) = 55, calls = 19
fib(20) = 6765, calls = 39
fib(30) = 832040, calls = 59
fib(50) = 12586269025, calls = 99
```

For n of 1 or more, the call count is $2n - 1$. The reason is this: each of fib(n − 1), …, fib(2) is computed once and makes two calls. There are n − 1 of them; adding the first call gives $1 + 2(n - 1) = 2n - 1$. For fib(30) the call count drops from 2,692,537 to 59. The time becomes $O(n)$. In exchange, an array of n + 1 cells is used: the extra memory is $O(n)$. Memory has been spent to gain time.

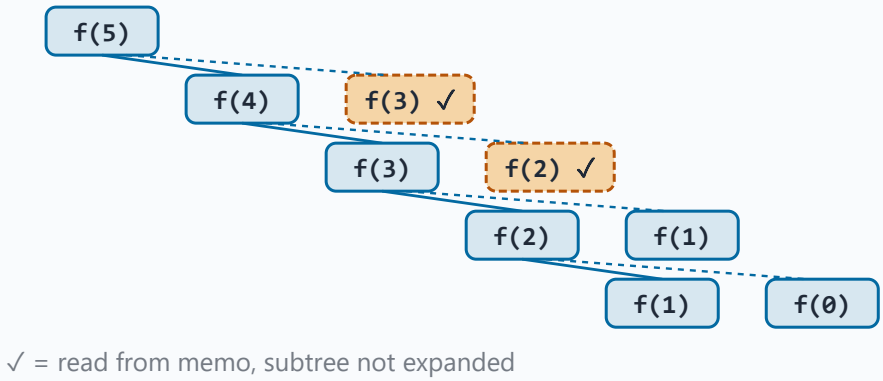


Figure 12.23: The calls of memoized fib(5). The tree has turned into a chain: 9 calls are made. The f(3) and f(2) marked ✓ are read from the memo; no new subtree is opened below them.

12.9.4 Tracing memoized fib(5)

Call no.	Frame	Work	memo
1	fib(5)	memo[5] == 0: to be computed	[0, 0, 0, 0, 0, 0]
2	fib(4)	memo[4] == 0: to be computed	[0, 0, 0, 0, 0, 0]
3	fib(3)	memo[3] == 0: to be computed	[0, 0, 0, 0, 0, 0]
4	fib(2)	memo[2] == 0: to be computed	[0, 0, 0, 0, 0, 0]
5	fib(1)	base case: return 1	[0, 0, 0, 0, 0, 0]
6	fib(0)	base case: return 0	[0, 0, 0, 0, 0, 0]
	fib(2) returns	memo[2] = 1 saved	[0, 0, 1, 0, 0, 0]
7	fib(1)	base case: return 1	[0, 0, 1, 0, 0, 0]
	fib(3) returns	memo[3] = 2 saved	[0, 0, 1, 2, 0, 0]
8	fib(2)	memo[2] = 1: from memo, return 1	[0, 0, 1, 2, 0, 0]
	fib(4) returns	memo[4] = 3 saved	[0, 0, 1, 2, 3, 0]
9	fib(3)	memo[3] = 2: from memo, return 2	[0, 0, 1, 2, 3, 0]
	fib(5) returns	memo[5] = 5 saved	[0, 0, 1, 2, 3, 5]

Table 12.20: The trace of memoized fib(5). The last column is the memo array after the work in that row.

Table 12.20 shows the nine calls and the memo filling up. The first four calls are fib(5), fib(4), fib(3) and fib(2). None of them is in the memo, and all call their left terms in order to be computed. fib(1) and fib(0) are base cases. fib(2) = 1 + 0 = 1 is computed and written to memo[2]. fib(3) calls fib(1) for its right term and finds 2; memo[3] becomes 2.

The effect of memoization shows in call 8. fib(4) calls fib(2) for its right term. In the plain version this call would open a new subtree. Here memo[2] = 1, and the value returns at once. In call 9, the fib(3) that fib(5) asks for also comes from the memo. In the end the memo array is [0, 0, 1, 2, 3, 5] and 5 is

printed. memo[0] and memo[1] are never used, because 0 and 1 are base cases.

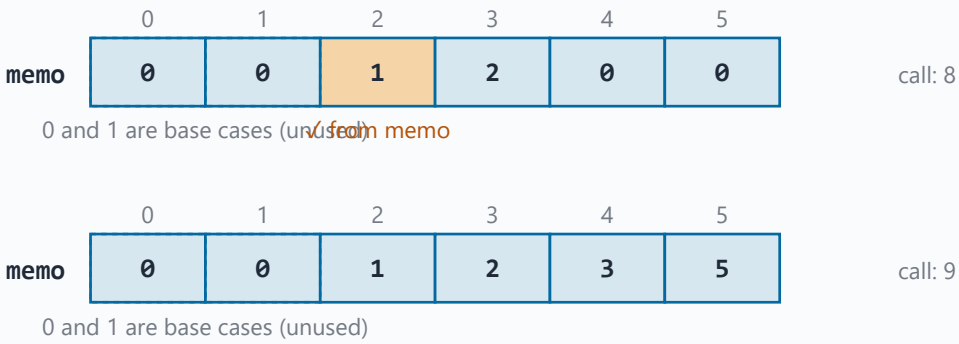


Figure 12.24: Two moments of the memo array. Top: call 8, where fib(2) is read from the memo and memo[2] = 1 and memo[3] = 2 are filled. Bottom: the final state, [0, 0, 1, 2, 3, 5], 9 calls in total.

Another version keeps the array as a `static` field instead of a parameter: `static long[] memo = new long[50];`. In this form the array is created once for the whole program, and later fib calls also use the earlier computations. An array of 50 cells is enough up to fib(49); fib(49) = 7,778,742,049 does not fit in an int but does fit in a long.

12.9.5 Combinations with memoization

Memoization is not specific to Fibonacci. It works in any recursion whose subproblems overlap. One of the exercises in the slides is the number of combinations. The number of ways to choose k items from n is written C(n, k) and satisfies the rule of Pascal's triangle: $C(n, k) = C(n - 1, k - 1) + C(n - 1, k)$. The rule is explained like this: a particular item is either chosen (k − 1 are chosen from the remaining n − 1 items) or not chosen (k are chosen from the remaining n − 1 items). The base cases are $C(n, 0) = 1$ and $C(n, n) = 1$: there is one way to choose no items and one way to choose all of them.

Example 12.2

The program below computes the combination count in plain and memoized form and compares the call counts of the two versions. The memoized version uses a two-dimensional `long` array; memo[n][k] holds the result for C(n, k).

```java
// Exercise from the slides: choose(n, k) = choose(n - 1, k - 1) + choose(n - 1, k)
public class Combinations {
    static long calls;

    static long choose(int n, int k) {
        calls++;
        if (k == 0 || k == n) return 1;
        return choose(n - 1, k - 1) + choose(n - 1, k);
    }

    static long chooseMemo(int n, int k, long[][] memo) {
        calls++;
        if (k == 0 || k == n) return 1;
        if (memo[n][k] != 0) return memo[n][k];
        memo[n][k] = chooseMemo(n - 1, k - 1, memo) + chooseMemo(n - 1, k, memo);
        return memo[n][k];
    }

    public static void main(String[] args) {
        int[][] g = {{4, 2}, {5, 2}, {10, 5}, {20, 10}, {30, 15}};
        for (int[] x : g) {
            calls = 0;
            long v = choose(x[0], x[1]);
            long c1 = calls;
            calls = 0;
            long w = chooseMemo(x[0], x[1], new long[x[0] + 1][x[1] + 1]);
            System.out.println("C(" + x[0] + ", " + x[1] + ") = " + v + " / " + w +
": plain " + c1 + " calls, memoized " + calls + " calls");
        }
    }
}
```

```
C(4, 2) = 6 / 6: plain 11 calls, memoized 9 calls
C(5, 2) = 10 / 10: plain 19 calls, memoized 13 calls
C(10, 5) = 252 / 252: plain 503 calls, memoized 51 calls
C(20, 10) = 184756 / 184756: plain 369511 calls, memoized 201 calls
C(30, 15) = 155117520 / 155117520: plain 310235039 calls, memoized 451 calls
```

The two versions give the same result: C(5, 2) = 10 and C(30, 15) = 155,117,520. The call count of the plain version is $2 \cdot C(n, k) - 1$, because the result is built by adding up the 1s at the leaves of the tree. For C(30, 15) this means more than 310 million calls. The memoized version computes each (n, k) pair once and finds the same result in 451 calls.

12.10 Tail recursion

12.10.1 The accumulator parameter

In recursive factorial, each frame leaves one job for the way back: multiplying the subresult by n. That is why the frame has to wait. The same computation can also be written so that the multiplication is done before the call. For this, the intermediate result is carried in a second parameter. This parameter is called the accumulator.

```
static long fact(int n, long acc) {
    if (n <= 1) return acc;
    return fact(n - 1, n * acc);
}
// fact(4, 1) → fact(3, 4)
//   → fact(2, 12) → fact(1, 24) → 24
```

fact(n, acc) computes the value *n! × acc*. In the first call, acc is given as 1: fact(4, 1) = 4!. In the recursive case the multiplication is done inside the call, while the argument is being computed: `n * acc`. When the call returns, no work remains; the result of the subcall is returned as it is. In the base case, acc carries the whole of the sought result.

Definition 12.8

Tail recursion is recursion in which the recursive call is the last operation of the method. After the call returns, no computation remains in the frame; the result of the subcall is returned unchanged.

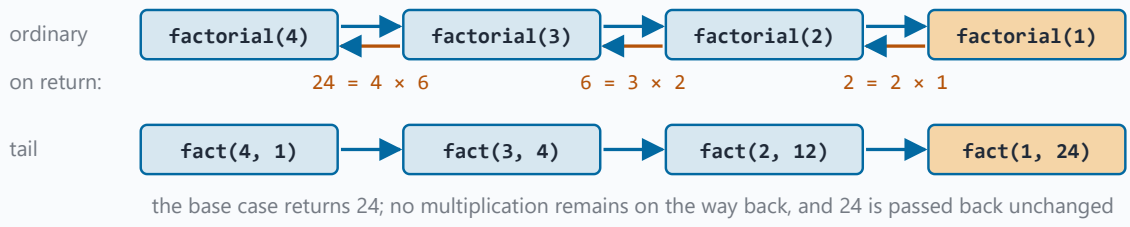


Figure 12.25: Ordinary and tail recursion. Left: in ordinary recursion the multiplications are done on the way back: 2 = 2 × 1, 6 = 3 × 2, 24 = 4 × 6. Right: in tail recursion the accumulator is carried forward; the base case returns 24 and no multiplication remains on the way back.

Since no work remains on the way back, there is no need to keep the old frame. Some languages and compilers recognize this situation. Instead of opening a new frame, they reuse the old one with the new arguments. This transformation is called tail call optimization. When the optimization is applied,

tail recursion uses as little memory as a loop: *O(1)*. The standard of languages such as Scheme requires it.

Java, however, does not guarantee this optimization. The Java compiler and the JDK 21 virtual machine do not turn tail calls into loops. Every call still opens a new frame. For this reason tail recursion does not give stack safety in Java: fact(100,000, 1) overflows just like any deep linear recursion. If you need deep tail recursion in Java, turning it into a loop is the programmer's job. The translation is easy: the parameters become loop variables, and the recursive call becomes an update of those variables.

12.10.2 Tracing fact(4, 1)

```
static long fact(int n, long acc) {
    if (n <= 1) return acc;
    return fact(n - 1, n * acc);
}
System.out.println(fact(4, 1)); // 24
```

The program prints 24. The version that was run also printed the results of fact(20, 1), fact(1, 1) and fact(0, 1):

```
24
2432902008176640000 1 1
```

Frame	n	acc	Work done before the call	Next step
fact(4, 1)	4	1	4 × 1 = 4	fact(3, 4)
fact(3, 4)	3	4	3 × 4 = 12	fact(2, 12)
fact(2, 12)	2	12	2 × 12 = 24	fact(1, 24)
fact(1, 24)	1	24	1 <= 1 → true	return 24

Table 12.21: The trace of fact(4, 1). The multiplications are done before each call.

Table 12.21 gives the chain: fact(4, 1) → fact(3, 4) → fact(2, 12) → fact(1, 24) → 24. The multiplications are *4 × 1 = 4*, *3 × 4 = 12* and *2 × 12 = 24*, and all of them are done on the way down. In ordinary factorial the multiplications were done on the way back in the order 2, 6, 24. Here the order of multiplication is 4, 12, 24, and on the way back only 24 is carried.

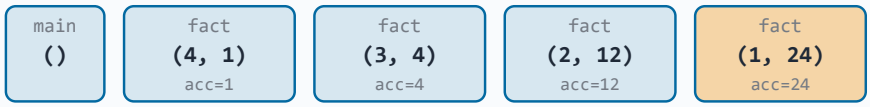


Figure 12.26: The moment the base case is reached in the call fact(4, 1). The four fact frames still sit on top of one another: Java does not remove the old frames. With main there are 5 frames.

Figure 12.26 shows the real situation in Java. The old frames have nothing left to wait for, but they stay on the stack. At the deepest moment there are 5 frames including main; this is the same as for ordinary factorial(4). Tail recursion does not reduce memory in Java; the extra memory is still *O(n)*. The benefit of the technique shows up in other languages. In Java it can be thought of as an intermediate step toward turning the recursion into a loop.

Common mistake

Starting the accumulator wrongly in the first call. In a method that accumulates a product, the accumulator starts at 1; in one that accumulates a sum, it starts at 0. The call fact(4, 0) returns 0, because it multiplies by 0 at every step. To prevent this mistake a one-parameter wrapper method is usually written: `static long factorial(int n) { return fact(n, 1); }`. Only this method is called from outside.

12.11 Overflow, edge cases and debugging

12.11.1 long overflow

Stack overflow and integer overflow are different things. In stack overflow, the calls do not fit in memory. In integer overflow, the computed number exceeds the largest value the type can hold. Factorial reaches this second kind very quickly. A long holds at most 9,223,372,036,854,775,807. 20! = 2,432,902,008,176,640,000 is below this limit. 21! = 51,090,942,171,709,440,000 is about five and a half times the limit.

Java does not throw an exception on integer overflow. The result silently wraps around: the part that exceeds the limit is discarded and the number comes back from the negative side. The program below ran the factorial method from 19 to 22:

```
public class Overflow {
    static long factorial(int n) {
        if (n < 0)
            throw new IllegalArgumentException();
        if (n <= 1) return 1;
        return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        for (int n = 19; n <= 22; n++)
            System.out.println("factorial(" + n + ") = " + factorial(n));
        System.out.println("Long.MAX_VALUE = " + Long.MAX_VALUE);
        System.out.println("21! (BigInteger) = " +
java.math.BigInteger.valueOf(21).multiply(new
java.math.BigInteger("2432902008176640000")));
        System.out.println("fib(92), fib(93) long: " + fib(92) + " " + fib(93));
        System.out.println("fastPow(2, 63) = " + fastPow(2, 63) + ", fastPow(2, 64) = " +
fastPow(2, 64));
        System.out.println("Integer.MIN_VALUE: Math.abs = " + Math.abs(Integer.MIN_VALUE) +
", Math.abs((long) ...) = " + Math.abs((long) Integer.MIN_VALUE));
    }

    static long fib(int n) {
        long a = 0, b = 1;
        for (int i = 0; i < n; i++) { long c = a + b; a = b; b = c; }
        return a;
    }

    static long fastPow(long base, int exp) {
        if (exp < 0) throw new IllegalArgumentException("exp < 0");
        if (exp == 0) return 1;
        long half = fastPow(base, exp / 2);
        if (exp % 2 == 0) return half * half;
        return half * half * base;
    }
}
```

```
factorial(19) = 121645100408832000
factorial(20) = 2432902008176640000
factorial(21) = -4249290049419214848
factorial(22) = -1250660718674968576
Long.MAX_VALUE = 9223372036854775807
21! (BigInteger) = 51090942171709440000
fib(92), fib(93) long: 7540113804746346429 -6246583658587674878
fastPow(2, 63) = -9223372036854775808, fastPow(2, 64) = 0
Integer.MIN_VALUE: Math.abs = -2147483648, Math.abs((long) ...) = 2147483648
```

factorial(21) returns a negative number. Because the program does not crash, the error can go unnoticed. The same problem exists in the other examples. fib(92) fits in a long, fib(93) does not and comes out negative. fastPow(2, 63) gives the smallest long value, and fastPow(2, 64) gives 0. If larger results are needed, the `java.math.BigInteger` class is used; its numbers can grow as long as memory allows. The 21! value in the program was also computed with BigInteger.

12.11.2 Negative exponents and other edge inputs

The call power(2, −1) was discussed in 12.4.1. The base case `n == 0` is never satisfied; n goes down through −1, −2, −3, … and the frames pile up. In the Depth program this call ended with a StackOverflowError after about 8 thousand calls. This number is smaller than the one for the forever method. Because the frame size depends on the method's parameters and on how the JVM happens to be running, the overflow point differs from method to method.

There are two remedies. The first is to check the input: if n < 0, an `IllegalArgumentException` is thrown; the factorial method already does this for negative input. The second is to make negative exponents well defined. The result type is changed to `double` and a negative exponent is converted to a positive one: $2^{-2} = 1 / 2^2 = 0.25$. A common version does this in one line: `if (n < 0) return 1.0 / power(b, -n);`. The negative branch runs only in the first call; in later calls the exponent is positive.

When testing a recursive method, the following inputs are tried:

1. **The base case itself:** factorial(0), factorial(1), sum(0), the empty string, a one-character string.
2. **One above the base case:** factorial(2), a two-character string. The simplest use of the recursive case is tested here.
3. **Invalid inputs:** a negative n, a negative exponent, a null string. The method must either throw a meaningful exception or return a defined value.
4. **Large inputs:** whether the result fits in the type (factorial(21)) and whether the depth fits on the stack (sum(100,000)).
5. **Special values:** 0 and Integer.MIN_VALUE in digit problems; the first and last elements of the array and a value not in the array when searching.

12.11.3 Debugging recursive code

If a recursive method gives a wrong result or never ends, the problem is usually in one of the two parts. Debugging is done in four steps:

1. **Show that the base case can be reached.** For every valid input, the chain of calls must end in the base case. This needs separate checking for jumping reductions (such as n − 2) and for negative inputs.
2. **Identify the shrinking measure.** Something must decrease on every call: n, the length of the string, the size of the range (hi − lo + 1) or the number of remaining elements (a.length − i).

3. **Put a small input in a table.** Write down the moments of call and return together with the depth and the returned values.
4. **Assess depth and repetition.** Does the maximum depth grow with the size of the input or with its logarithm? Is the same argument seen more than once in the recursion tree?

For the third step, temporary print lines can be added to the method. The depth is kept in a counter, and each line is printed indented by the depth. The nesting of the calls then shows in the output. Table 12.22 is such a trace for sum(2).

Depth	Event	Value
0	sum(2) called	
1	sum(1) called	
2	sum(0) called	
2	sum(0) returned	0
1	sum(1) returned	1
0	sum(2) returned	3

Table 12.22: An indented trace of calls and returns for sum(2). Depth 0 is the first call.

The table shows the opening and closing of a call at the same indentation level. sum(2) opens at depth 0 and closes last. In between, sum(1) and sum(0) open and close. The returned values are 0, 1 and 3. When comparing with the expected value, look for the first call that returns a wrong value. The error is in that call's own lines, because all the subresults it uses are correct.

How deep a method goes and how many calls it makes can also be measured with a version that has counters. All the call counts and depths given in these notes were measured this way. The counter is a `static` field incremented on entry to the method. For depth, a second field that is incremented on entry and decremented on exit is enough, together with a third field that keeps the largest value seen.

The Recursion Visualizer simulator shows these traces interactively. In the simulator, the calls of factorial, fib, power and hanoi open and close step by step. The call stack and the recursion tree can be followed at the same time. For Fibonacci, memoization can be switched on and off to compare the call counts of the two versions.

12.12 Summary

- Recursion is the technique in which a method reduces a problem to a smaller instance of the same problem and calls itself. Every recursive method has two parts: the base case, where the result is known directly, and the recursive case, which shrinks the problem and calls the method itself.
- The problem must shrink measurably on every call and must reach the base case. When the input shrinks in jumps, or for negative inputs, it is safer to write the base case with `<=` instead of `==`.

- Each call opens a stack frame that holds its parameters, local variables and return point. Frames close in the reverse of the order in which they were opened. Work placed before the recursive call is done on the way down; work placed after it is done on the way back, in reverse order.
- Linear recursion (factorial, sum, plain power) opens n frames: time and extra stack are *O(n)*. In problems that shrink substantially at every step, such as digit sum, fast power and binary search, the depth is logarithmic.
- Divide and conquer splits a problem into parts, solves the parts recursively and combines them. Merge sort runs in *O(n log n)* time with *O(n)* extra memory and a stack depth of *O(log n)*.
- Methods in which each call makes two calls form a recursion tree. The Towers of Hanoi takes $2^n - 1$ moves, and subset generation produces 2^n subsets. In these problems the exponential work comes from the problem itself; the stack depth is n + 1.
- Recursion that has no base case, or that goes very deep, fills the stack and ends with a StackOverflowError. On JDK 21 the default stack could carry a few tens of thousands of calls.
- For simple linear repetition, a loop runs in a single frame with *O(1)* extra stack and does not overflow. For self-similar problems, recursion is close to the definition and readable. The choice is made on correctness, readability and resource use.
- Direct Fibonacci solves the same subproblems again and again. The call count is $2 \cdot fib(n + 1) - 1$ and grows exponentially: 2,692,537 calls for fib(30).
- Memoization removes the repetition by saving subproblem results. Memoized fib(n) makes $2n - 1$ calls; the time drops to *O(n)*, at the price of *O(n)* memory.
- In tail recursion the call is the last thing the method does, and the intermediate result is carried in an accumulator parameter. Java does not optimize tail calls; the frames still pile up and the extra memory remains *O(n)*.
- Integer overflow is different from stack overflow, and it is silent: long factorial(21) returns a negative number. Recursive methods are tested with the base case, one above the base case, invalid input and large input.

12.13 Key terms

recursion base case recursive case call stack stack frame recursion depth recursion tree helper method throw (an exception) palindrome greatest common divisor Euclid's algorithm substring divide and conquer binary search Towers of Hanoi subset backtracking stack overflow infinite recursion iterative version time complexity exponential growth Fibonacci sequence memoization tail recursion integer overflow

12.14 Test Yourself

1. In the call sum(5) (sum(n) = n + sum(n − 1), sum(0) = 0), how many times is the sum method called? How many frames does the stack hold at its deepest moment, counting main? What is the result?

   Answer

   sum(5), sum(4), sum(3), sum(2), sum(1) and sum(0) are called: 6 calls. When sum(0) is called, no call has returned yet; with main there are 7 frames on the stack. The result is *5 + 4 + 3 + 2 + 1 + 0 = 15*.

2. What is the chain of calls for gcd(21, 13)? How many calls are made and what is the result?

   Answer

   gcd(21, 13) → gcd(13, 8) → gcd(8, 5) → gcd(5, 3) → gcd(3, 2) → gcd(2, 1) → gcd(1, 0). In total 7 calls are made and the result is 1. 21 and 13 are consecutive Fibonacci numbers. For this reason the remainders also shrink one step at a time as Fibonacci numbers; this is the slowest case for Euclid's algorithm.

3. In the call fastPow(2, 13), what values does the exponent pass through? How many calls are made? Which frames multiply by the base once more on the way back? How many calls would plain power(2, 13) make?

   Answer

   The exponent is 13, 6, 3, 1 and 0: 5 calls. On the way back, the frames with an odd exponent (1, 3 and 13) multiply the result by 2 once more. The values are 1, 2, 8, 64 and 8192 in turn: *1·1·2 = 2*, *2·2·2 = 8*, *8·8 = 64*, *64·64·2 = 8192*. The plain method would make *n + 1 = 14* calls.

4. How many moves are needed for 6 disks in the Towers of Hanoi? How many times is the hanoi method given in the notes called (counting the hanoi(0) calls)?

   Answer

   The number of moves is $2^6 - 1 = 63$. The number of calls is $2^7 - 1 = 127$: the number of calls doubles at each level of the recursion tree, and there are 7 levels (from n = 6 down to n = 0). All the leaves are hanoi(0) calls, which make no move.

5. When fib(10) is computed, how many calls do the plain recursive version and the memoized

version make? Which formulas give the numbers?

Answer

The plain version makes $2 \cdot fib(11) - 1 = 2 \cdot 89 - 1 = 177$ calls. The memoized version makes $2 \cdot 10 - 1 = 19$ calls. The result is fib(10) = 55 in both.

6. The subsets method given in the notes is called with the array {1, 2, 3}. How many lines are printed? What is the 4th line? How many times is the method called?

Answer

$2^3 = 8$ lines are printed: [1, 2, 3], [1, 2], [1, 3], [1], [2, 3], [2], [3], []. The 4th line is [1]: 1 is included, and 2 and 3 are excluded. The number of calls is $2^4 - 1 = 15$.

7. On the array a = {2, 5, 8, 12, 16, 23, 38, 56}, how many calls does search(a, 5, 0, 7) make, and what does it return?

Answer

In the first call mid = 3 and a[3] = 12; since $12 > 5$, search(a, 5, 0, 2) is called. In this call mid = 1 and a[1] = 5: the value has been found. 2 calls are made and 1 is returned.

8. The tail-recursive version of the sum from 1 to n is: `static int sumTail(int n, int acc) { if (n == 0) return acc; return sumTail(n - 1, acc + n); }`. What is the chain of calls for sumTail(4, 0)? In Java, how many frames are on the stack at the deepest moment, counting main?

Answer

sumTail(4, 0) → sumTail(3, 4) → sumTail(2, 7) → sumTail(1, 9) → sumTail(0, 10) → 10. The additions are done before the calls and no work remains on the way back. Because Java does not optimize tail calls, all five sumTail frames are on the stack; with main there are 6 frames.

9. A method count(s, c, i) counts how many times a character occurs in a string, starting from index i. What should the base case and the recursive case be? How many calls does count("kayak", 'a', 0) make, and what does it return?

Answer

The base case is `i == s.length()`: no characters are left to look at, and 0 is returned. In the recursive case, 1 is taken if s.charAt(i) equals c and 0 otherwise, and the value is added to count(s, c, i + 1). For “kayak”, 6 calls are made, for i = 0 to 5. The 'a' characters are at indices 1 and 3; the result is 2.

12.15 Further reading

- R. Sedgewick and K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), Section 2.3: recursion is presented through Euclid's algorithm, the Towers of Hanoi, Gray codes and recursive graphics; the section also explains why direct Fibonacci is slow and how memoization speeds it up. The online version is at introcs.cs.princeton.edu.
- E. Roberts, *Thinking Recursively with Java* (Wiley, 2006): a book devoted entirely to thinking recursively. It builds, step by step, the habit of assuming the small problem is solved and constructing the big problem from it.
- A. B. Downey and C. Mayfield, *Think Java* (2nd ed., O'Reilly, 2019): recursive methods and stack diagrams are explained in plain language for people new to programming; the online version of the book is free.
- Oracle, *Java SE 21 API*: the official description of the StackOverflowError class; it states that the error is thrown when an application recurses too deeply.

12.16 References

1. E. Roberts, *Thinking Recursively with Java*, Wiley, 2006.
2. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 2.3 Recursion.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
4. Y. D. Liang, *Introduction to Java Programming and Data Structures*, 13th ed., Pearson, 2024.
5. Oracle, *Java SE 21 API: StackOverflowError*.
6. A. B. Downey, C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019.

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

absolute path 7
The full location of a file or folder starting from the root of a drive, such as C:\Users\Ann\Documents.

absolute reference 9
A cell reference that does not change when the formula is copied, written with dollar signs ($A$1).

absolute value 16
The distance of a number from zero, ignoring its sign; |-5| is 5 (Math.abs in Java).

abstraction 4
Hiding unnecessary details so that only what is needed to use something is visible.

accept / reject 8
Review commands that approve or discard a tracked change in a Word document.

access (public) 14
The visibility level (such as public) that determines which code may use a class or member.

access point 3
A device that connects wireless devices to a wired network.

accessibility 5
Designing products so that people with disabilities, such as vision or hearing impairments, can use them.

accessibility checker 8
A Word tool that checks a document for content that is hard for people with disabilities to use.

accumulator 13
A variable that collects a running total or result as a loop repeats.

accumulator (ACC) 4
The CPU register that holds the intermediate results of arithmetic and logic operations.

acknowledgment (ACK) 2
A message (ACK) sent by a receiver to confirm that data arrived correctly.

action button 10
A PowerPoint shape that performs an action, such as moving to the next slide, when clicked.

active cell 9
The worksheet cell that is currently selected and in which typing takes effect.

actual argument 18
The actual value or expression supplied for a parameter when a method is called.

actuator 1
A device, such as a motor or valve, that turns a control signal into physical action.

ADC / DAC 3
An analog-to-digital converter turns real-world signals into numbers; a digital-to-analog converter does the reverse.

address bar 7
The box at the top of a file manager window that shows the location of the current folder.

address space 6
The range of memory addresses that a process or the processor can use.

adjacent 12
Next to each other; adjacent cells or array elements sit side by side.

administrator 7
A user with the authority to change system settings and install software.

administrator account 6
A user account that has full control over the computer and its settings.

agile 11
A software development approach that works in short cycles, delivers frequently and welcomes change.

algorithm 5
A finite, ordered sequence of clear steps that solves a problem.

algorithm analysis 12
The study of how much time and memory an algorithm needs as the input grows.

algorithm design 5
Planning the steps and structure of a solution before it is written as a program.

align / distribute 10
PowerPoint commands that line up selected objects or space them evenly.

alignment 8
The placement of text relative to the margins: left, center, right or justified.

alt text 8
A short text description of an image that screen readers read aloud.

alternating / direct current 3
Two forms of electric current: alternating current (AC) changes direction periodically, direct current (DC) flows one way.

analog 1
Representing information with continuously varying values rather than discrete digits.

AND / OR / NOT / XOR 4
The basic logic operations: AND is true if both inputs are true, OR if at least one is, NOT inverts, XOR if exactly one is.

animation 10
A visual effect that makes an object on a slide move or change.

Animation Painter 10
A PowerPoint tool that copies the animation of one object to another.

animation pane 10
The PowerPoint pane that lists and orders the animations on a slide.

annotation (flowchart) 13
A note attached to a flowchart symbol that explains a step without being part of the flow.

application 1
A program written to perform a task for the user, such as a word processor or browser.

application / app 7
A program that a user runs to do a specific job, such as a browser or editor.

application / transport / internet / link layer 2
The four layers of the TCP/IP model, from the user's programs down to the physical link.

application programming interface (API) 11
A set of rules and functions through which one program can use the services of another.

application software 6
Programs that let users do tasks such as writing, calculating or browsing, as opposed to system software.

approximate match 9
A lookup that finds the closest value that does not exceed the search key (VLOOKUP with TRUE).

architecture 4
The overall design of a system, showing its parts and how they fit together.

archive / compress 7
To gather files into one file or reduce their size, for example into a ZIP archive.

argument 18
A value passed to a method when it is called.

argument (Excel) 9
A value, cell or range that is passed to an Excel function, such as the range in SUM(A1:A5).

arithmetic logic unit (ALU) 1
The CPU part (ALU) that carries out arithmetic and logic operations.

arithmetic operation 1
An operation such as addition, subtraction, multiplication or division.

Armstrong number 17
A number equal to the sum of its digits each raised to the power of the number of digits, such as 153.

array 19
An ordered collection of elements of the same type, accessed by index.

array initializer 19
The list of values in braces that sets the elements of an array when it is created, such as {1, 2, 3}.

arrow form (->) 16
The lambda syntax in which parameters and the body are separated by an arrow.

artificial intelligence 1
The field of making computers perform tasks that normally need human intelligence.

artificial intelligence (AI) 2
The ability of computer systems to learn, reason and decide in ways that resemble human intelligence.

ASCII 4
A character encoding that represents English letters, digits and symbols with 7-bit codes.

aspect ratio 3
The ratio of an image's or screen's width to its height, such as 16:9.

assembler 11
A program that translates assembly language into machine code.

assembly language 4
A low-level language whose instructions map almost one-to-one to machine instructions.

assignment 15
Storing a value in a variable, written with the = operator.

assignment statement 13
A statement that stores the value of an expression in a variable.

asymmetric encryption 2
Encryption that uses a pair of keys, a public key to encrypt and a private key to decrypt.

attachment 2
A file sent along with an e-mail message.

attributes 7
Properties of a file, such as read-only or hidden, kept by the file system.

audience 10
The people who watch and listen to a presentation.

authentication 6
Proving that a user or device is who it claims to be, for example with a password.

authentication / verification 7
Checking a user's identity or the correctness of information before allowing access.

authoritative server 2
A DNS server that holds the official records for a domain and gives final answers about it.

authorization 6
Deciding what an authenticated user is allowed to do.

AutoCorrect 8
A Word feature that fixes common typing mistakes automatically as you type.

AutoFill 9
An Excel feature that continues a pattern or series into adjacent cells.

automation 1
Using machines or software to carry out tasks with little human involvement.

AutoRecover 8
A Word feature that recovers unsaved work after a crash.

AutoSave 8
A Microsoft 365 feature that saves changes to a cloud-stored file continuously.

availability 2
The property that a system and its data can be reached whenever they are needed.

axis 10
One of the reference lines of a chart, such as the horizontal category axis or the vertical value axis.

B

backlight 3
The light source behind an LCD screen that makes the image visible.

backtracking 21
A technique that tries a choice, and when it fails goes back and tries another.

backup 3
A copy of data kept separately so that it can be restored after loss or damage.

backward compatible 3
Able to work with older versions of hardware or software.

bandwidth 3
The maximum amount of data a connection can carry per unit of time.

bar / column chart 10
A chart that shows values as rectangular bars or columns whose length represents the value.

base (radix) 4
The number of distinct digits in a number system, such as 2 for binary and 10 for decimal.

base and limit registers 6
Two registers that define the memory range a process may access.

base case 21
The condition in a recursive method that stops the recursion without another call.

best case 12
The input for which an algorithm does the least work.

bias (IEEE 754 exponent) 4
A constant added to the exponent in IEEE 754 so that it can be stored as an unsigned number.

bibliography 8
A list of the sources cited in a document, usually placed at the end.

big data 1
Data sets so large or fast-changing that ordinary tools cannot process them easily.

Big O notation 12
A notation that describes how an algorithm's running time grows with the input size, such as O(n).

binary 11
Related to a base-2 system that uses only the digits 0 and 1.

binary (number system) 1
The base-2 number system, which uses only the digits 0 and 1.

binary digit 1
A single position in a binary number that holds 0 or 1; shortened to bit.

binary search 12
A search that halves a sorted list at each step by comparing with the middle element.

biometric 7
Based on a physical characteristic of a person, such as a fingerprint or face.

BIOS / UEFI 3
The firmware that tests the hardware and starts the operating system when the computer powers on; UEFI is its modern replacement.

bit 1
The smallest unit of information, with the value 0 or 1.

bit depth 1
The number of bits used to represent each sample or pixel color.

bit mask 4
A pattern of bits used with AND, OR or XOR to select, set or clear specific bits.

bit shift (<<, >>) 4
An operation that moves the bits of a value left or right by a given number of positions.

bitwise operation 4
An operation applied to the individual bits of values, such as AND, OR, XOR and NOT.

block 7
A group of statements treated as one unit, written between curly braces in Java.

bold / italic / underline 8
Basic character formatting commands that make text heavier, slanted or underlined.

bookmark 8
A named location in a document to which a link can jump.

boolean 16
A data type with only two values, true and false.

boot 6
To start a computer and load the operating system into memory.

boot / booting 6
The process of starting a computer and loading the operating system.

boot loader 6
A small program that loads the operating system kernel into memory.

boot order 6
The order in which the firmware checks devices for an operating system to start.

bootloader 6
A small program that loads the operating system kernel into memory.

border 9
The line around the edge of a cell, table or object.

border and shading 8
Word formatting that draws lines around text and fills its background.

boundary (edge) case 14
An input at the edge of the allowed range, such as the first or last array index.

boundary value 13
A value at the limit of a range, used when testing, such as the smallest and largest allowed input.

branch 13
A point in a flowchart or program where the flow splits into different paths.

branch (if/else) 18
One of the alternative paths of a conditional statement, taken when its condition is true or false.

branch (instruction) 4
A machine instruction that changes the order of execution by jumping to another address.

branching 13
Choosing between paths of execution depending on a condition.

break (statement) 17
A statement that ends the nearest enclosing loop or switch immediately.

breakpoint 11
A marker that stops a running program so the programmer can inspect it.

brightness / contrast 3
Settings that control how light or dark an image is and how large the difference between its light and dark areas is.

bubble sort 12
A sorting algorithm that repeatedly swaps adjacent elements that are out of order.

buffer 6
A memory area that temporarily holds data while it is moved between two parts of a system.

buffer (input buffer) 20
A memory area that holds typed characters until the program reads them.

bulleted list 8
A list in which each item is marked with a bullet.

bus (data / address / control) 4
The three groups of wires that carry data, addresses and control signals between the CPU, memory and I/O.

byte 1
A group of 8 bits, the basic unit of memory size.

byte (unit) 15
A unit of storage made of 8 bits.

bytecode 14
The platform-independent intermediate code that the Java compiler produces and the JVM runs.

bytecode verifier 14
A JVM component that checks bytecode for safety before it runs.

C

cache 2
A small, fast memory that keeps recently used data close to the processor or client.

cache (memory) 3
A small, fast memory between the CPU and main memory that holds recently used data.

cache hit / miss 3
A hit occurs when requested data is found in the cache; a miss occurs when it is not.

cache level (L1 / L2 / L3) 3
One of the layers of cache, with L1 the smallest and fastest and L3 the largest and slowest.

cache line (block) 3
The unit of data that is moved between memory and the cache at one time.

Caesar cipher 20
A simple cipher that shifts every letter by a fixed number of positions in the alphabet.

call 12
A statement that runs a method or function.

call stack 21
The stack that stores the active method calls and their local data.

call stack / stack frame 21
The stack of active method calls; each call occupies one frame holding its parameters and local variables.

capacity 3
The amount of data a storage device can hold.

caption 8
A line of text placed next to a figure or table to describe it.

captions / subtitles 10
Text shown on screen that gives the words spoken in a video or audio.

carry 4
A digit passed to the next higher position when an addition result exceeds the digit range.

case label 16
A label in a switch statement that marks a value to be matched.

case-sensitive 14
Treating uppercase and lowercase letters as different, so Name and name are not the same.

cast (explicit conversion) 15
Converting a value to another type with a type name in parentheses, such as (int) 3.7.

cell 8
The box where a row and a column intersect in a table or worksheet.

cell (array slot) 19
One position of an array that holds a single value.

cell reference 9
The address of a cell, made of its column letter and row number, such as B3.

central processing unit (CPU) 3
The main chip of a computer that fetches, decodes and executes instructions.

certificate 2
A digital document that binds a public key to an owner and is signed by a trusted authority.

channel 2
A path through which data travels, such as a wireless frequency band.

channel width (Wi-Fi) 3
The width of the frequency band a Wi-Fi channel uses, for example 20 or 80 MHz.

character 4
A single letter, digit or symbol.

character encoding 14
A rule that maps characters to numeric codes, such as UTF-8 or UTF-16.

character set 1
The collection of characters that a system can represent.

chart 10
A graphic that presents worksheet or table data visually.

chip 3
A small piece of semiconductor material that holds an integrated circuit.

chipset 3
The set of chips on a motherboard that connects the CPU with memory and devices.

circular wait 6
A condition for deadlock in which processes wait for each other in a cycle.

citation 8
A reference in the text to a source that supports a statement.

citation style 8
A standard format for citations and the bibliography, such as APA or IEEE.

class 14
A blueprint that defines the fields and methods of objects.

class declaration 14
The code that defines a class, with its name, fields and methods.

class library 14
A collection of ready-made classes that programs can reuse.

class loader 14
A JVM component that finds classes and loads them into memory when needed.

client 1
A computer or program that requests services from a server.

clipboard 8
A temporary storage area for items that are copied or cut.

clock cycle 3
One tick of the processor clock, the basic unit of time for executing instructions.

clock speed 3
How many clock cycles the processor performs per second, measured in hertz.

closed source / proprietary 6
Software whose source code is not published and whose use is restricted by its owner.

cloud 8
Servers and services reached over the internet instead of on a local computer.

cloud computing 1
Using computing resources such as servers, storage and software over the internet on demand.

co-authoring 10
Several people writing the same document at the same time.

code 12
A set of instructions written in a programming language.

code editor 11
A text editor made for writing programs, with features such as highlighting and completion.

code generation 11
The compiler stage that produces machine code or bytecode.

code optimization 11
Improving generated code so that it runs faster or uses less memory.

code point 4
The number assigned to a character in Unicode.

code tracing 18
Following a program by hand, step by step, recording how variable values change.

code unit 4
The unit of an encoding, such as 8 bits in UTF-8 or 16 bits in UTF-16.

code unit (UTF-16) 20
A 16-bit unit of UTF-16; a char in Java is one such unit.

coding 11
Writing the solution in a programming language.

color accuracy 3
How closely the colors shown by a display match the real colors.

color depth 1
The number of bits used to represent the color of each pixel.

color scales 9
A conditional format that shades cells in a gradient according to their values.

column 8
A vertical line of cells in a table or worksheet.

column / bar chart 9
A chart that shows values as vertical bars.

command 8
An instruction given to a program, such as a menu item or ribbon button.

command bar 7
The toolbar in an application or window that holds its commands.

command line 14
A text interface in which the user types commands.

command prompt 6
The Windows program for typing text commands (cmd).

command queue 3
A queue of commands waiting to be executed by a device such as a disk controller.

command-line argument 14
A value given to a program when it is started from the command line.

command-line interface (CLI) 6
A text-based interface for interacting with the operating system or a program.

comment 8
A note in the source code or document that the computer ignores but readers see.

commit (version control) 11
To save a snapshot of changes in a version control system.

comparison 1
Deciding whether one value is equal to, less than or greater than another.

comparison operator 15
An operator such as ==, !=, < or > that compares two values and gives true or false.

compatibility 3
The ability of hardware or software to work together with other products.

Compatibility Checker 10
A PowerPoint tool that finds features not supported by earlier versions of the program.

compile 11
To translate source code into machine code or bytecode.

compile / compilation 11
The translation of source code into machine code or bytecode.

compile error 14
An error found by the compiler because the code breaks the language rules.

compile-time error 15
An error found when the compiler checks the source code, before the program runs.

compiled language 11
A language whose programs are translated entirely before they run, such as C and Java.

compiler 11
A program that translates source code into machine code or bytecode.

compound assignment 15
An assignment operator that combines an operation with assignment, such as += and *=.

compound condition 13
A condition made of simple conditions joined by AND, OR or NOT.

compress (media) 10
To reduce the size of images, audio or video files, sometimes with some loss of quality.

computability 5
The question of whether a problem can be solved by an algorithm at all.

computation 5
Processing data according to rules to produce a result.

computational thinking 5
A way of solving problems by decomposition, pattern recognition, abstraction and algorithms.

computer 1
A machine that accepts data, processes it according to stored instructions and produces results.

computer case 3
The housing that holds a computer's internal components.

computer network 2
A group of computers and devices connected so that they can share data and resources.

computer science 5
The study of computation, algorithms, data and information systems.

concatenation (Excel &) 9
Joining two text values into one with the & operator or the CONCAT function.

concurrency 6
The ability of a system to manage several tasks whose execution overlaps in time.

condition 1
A statement that is either true or false and controls a decision or loop.

conditional (ternary) operator 16
The ?: operator that picks one of two values depending on a condition.

conditional flow (selection) 13
A control structure that chooses which path to follow based on a condition.

conditional formatting 9
A feature that changes the look of cells automatically when they meet a rule.

confidentiality 2
Keeping information from being seen by people who are not authorized.

connector (flowchart, shape) 13
A flowchart shape, usually a circle, that joins parts of a flowchart or continues on another page.

connector (hardware) 3
A plug or socket that joins a cable or card to a device.

console 14
A text window for entering input and showing the output of a program.

constant 9
A cell or value whose content does not change.

constant / literal 15
A fixed value written directly in the code, such as 42 or "hello".

constant expression 20
An expression the compiler can evaluate at compile time, such as a case label.

constant folding 11
An optimization in which the compiler evaluates constant expressions in advance.

constant pool 14
The part of a class file that holds constants and names used by the class.

constraint 12
A rule that a solution must satisfy.

container 6
A package that holds an application and everything it needs and runs isolated from others.

container engine 6
The software that creates and runs containers, such as Docker.

content delivery network (CDN) 2
A network of distributed servers that delivers web content from a location close to the user.

content vs reference comparison 20
Comparing the contents of two objects (equals) versus comparing whether they are the same reference (==).

context menu 7
The menu that appears when you right-click an item and offers actions for it.

context switch 6
Switching the CPU from one process to another by saving and restoring their state.

contextual tab 8
A ribbon tab that appears only when a certain object, such as a picture or table, is selected.

continue 13
A statement that skips the rest of the current loop iteration and goes on to the next one.

contract (method contract) 18
The agreement that states what a method requires and what it guarantees.

contrast 10
The difference in brightness between the light and dark parts of an image.

control character 4
A non-printing character that controls text, such as newline or tab.

control flow 16
The order in which the statements of a program are executed.

control structure 16
A structure such as a conditional or loop that controls the order of execution.

control unit 4
The CPU part that directs the other parts by decoding instructions and issuing control signals.

controller 7
A hardware component or program that manages a device and handles its communication.

controller (memory / keyboard / SSD) 3
A chip or circuit that manages a device such as memory, keyboard or SSD.

cookie 2
A small piece of data that a website stores in the browser to recognize the user later.

cooling 3
The methods, such as fans and heat sinks, that remove heat from a computer.

Copilot 10
Microsoft's AI assistant that helps create content in Office applications.

copy 19
A duplicate of an array or object.

copy / cut / paste 7
The commands for duplicating, moving and pasting selected content.

copy-on-write 6
A technique in which memory pages are shared until one process writes, and only then is a copy made.

copyleft 6
A licensing approach that requires derived works to carry the same free license.

core (CPU) 3
An independent processing unit inside a CPU.

correctness 12
Whether an algorithm gives the right result for every valid input.

cost 12
The amount of resources, such as time or memory, that an algorithm uses.

counter 13
A variable that counts how many times something has happened.

counter-controlled loop 13
A loop that repeats a fixed number of times controlled by a counter variable.

CPI (cycles per instruction) 4
The average number of clock cycles a processor needs to execute one instruction.

CPU 7
The central processing unit, the main processor of a computer.

criteria 9
A condition that cells must meet to be counted or selected by functions such as COUNTIF.

crop 8
To cut away the outer parts of an image.

cross-reference 8
A reference in a Word document to another place, such as a figure or heading.

curly braces 14
The symbols { and } that enclose a block in Java.

current (working) directory 6
The folder in which a program or the command line is currently working.

current folder 7
The folder whose contents are currently being shown or used.

cursor 3
The marker that shows the position of the pointer on the screen.

cursor (text) 8
The blinking mark that shows where typed text will appear.

custom show 10
A PowerPoint show that includes only the chosen slides, for a particular audience.

D

dangling else 16
The ambiguity in nested if statements about which if an else belongs to; it belongs to the nearest one.

data 1
Raw facts, numbers or symbols that have not yet been processed into information.

data bars 9
A conditional format that draws a bar in a cell whose length is proportional to its value.

data entry 9
Typing values into cells or fields.

data life cycle 1
The stages data passes through, from creation and storage to use and disposal.

data series 9
A set of related values plotted in a chart, such as one row or column.

data source 8
The file or database that supplies the data for a mail merge.

data store (flowchart symbol) 13
A flowchart symbol that shows data being stored or retrieved.

data type 4
The kind of values a variable can hold and the operations allowed on them.

data type (Excel) 9
The kind of value a cell holds, such as text, number or date.

data validation 9
An Excel feature that limits what can be entered in a cell.

database 1
An organized collection of data that can be stored, searched and updated.

date serial number 9
The number that Excel uses to store a date, counted in days from a starting date.

De Morgan's laws 16
Two rules for negating compound conditions: !(a && b) equals !a || !b, and !(a || b) equals !a && !b.

deadline 6
The time by which a task must be finished.

deadlock 6
A situation in which processes wait forever for each other's resources.

debugger 11
A tool that runs a program step by step so errors can be found.

debugging 6
Finding and fixing errors in a program.

decimal 4
Related to the base-10 system with the digits 0 to 9.

decimal (base 10) 17
The base-10 number system, which uses the digits 0 to 9.

decimal (floating-point) 20
A number with a fractional part, such as 3.14, stored in floating-point form.

decimal separator 15
The character, a point or comma, that separates the integer and fractional parts of a number.

decision 13
A step in a flowchart or program where one of several paths is chosen.

decision structure 16
A structure, such as if or switch, that selects what to do depending on a condition.

decision symbol 13
The diamond shape in a flowchart that asks a yes/no question.

declaration 14
A statement that introduces the name and type of a variable, method or class.

declaration statement 14
A statement that introduces a variable and its type, such as int x;.

declarative knowledge 5
Knowledge of facts and what things are, as opposed to knowing how to do something.

decomposition 5
Breaking a large problem into smaller, easier parts.

decryption 2
Converting encrypted data back into readable form.

default app 7
The program that Windows uses to open a file type unless told otherwise.

default gateway 2
The router address that a device uses to send traffic to other networks.

default port 2
The standard port number of a service, such as 80 for HTTP and 443 for HTTPS.

default route 2
The route that a router uses when no other route matches a destination.

default value 19
The value that a variable or element has when none is given, such as 0 for an int array element.

definiteness 12
The property that each step of an algorithm is clearly and precisely stated.

defragmentation 7
Reorganizing the fragments of files on a disk so that each file is stored contiguously.

delay 10
The time to wait before an animation or action starts.

delimiter / separator 20
A character, such as a comma or space, that separates parts of a text.

dependency 6
A resource or software component that something else needs in order to work.

deployment 11
Making finished software available for use.

deprecated 20
Marked as outdated; still works but should no longer be used.

design 11
The phase of planning how the software will meet the requirements.

Designer 10
A PowerPoint feature that suggests layouts and design ideas for slides.

desktop 7
The main screen of the operating system, where icons and windows are shown.

desktop / laptop 1
A computer for a fixed place or a portable computer.

device 3
A piece of hardware that can be attached to a computer.

device driver 6
Software that lets the operating system communicate with a hardware device.

digit 12
One position of a number, such as the tens digit.

digit (symbol) 17
One of the symbols used to write numbers, such as 0 to 9.

digit / place value 1
A symbol in a number and the value that its position gives it.

digital 1
Representing information with discrete values, usually 0 and 1.

digital footprint 2
The trail of data that a person leaves behind through online activity.

digital signature 2
An electronic signature that proves who sent a message and that it has not been changed.

DIMM / PCIe / M.2 / SATA / NVMe 3
Names of memory and storage interfaces and form factors found in computers.

direct formatting 8
Formatting applied straight to selected text rather than through a style.

disk 7
A device that stores data permanently, such as a hard disk or SSD.

display panel (IPS / VA / TN / OLED / LCD) 3
The screen technologies used in monitors, such as IPS, VA, TN, OLED and LCD.

divide and conquer 21
A strategy that splits a problem into smaller subproblems, solves them and combines the results.

dividend 15
The number that is divided in a division.

divisor 12
The number by which the dividend is divided.

DNS (Domain Name System) 2
The system that translates domain names such as example.com into IP addresses.

DNS resolution 2
The process of finding the IP address that belongs to a domain name.

DNS resolver 2
A server or program that asks DNS servers on behalf of a client to find an address.

document 8
A file created in a word processor, such as a letter or report.

Document Inspector 10
A PowerPoint tool that finds and removes hidden data and personal information.

domain name 2
The readable name of a website, such as example.com.

domain-specific language 11
A programming language designed for one particular area, such as SQL for databases.

dotted decimal notation 2
The way IPv4 addresses are written as four decimal numbers separated by dots.

DPI 3
Dots per inch; a measure of printing or display resolution.

drag and drop 7
Moving an item by holding it with the mouse and releasing it at another place.

drive 7
A storage unit that appears with a letter in the file system, such as C:.

drive letter 6
The letter that Windows assigns to a drive, such as C: or D:.

drop-down list 9
A control that shows a list of choices when clicked.

duplex printing 8
Printing on both sides of the paper.

duration 10
How long an animation or slide lasts.

dynamic typing 11
A system in which types are checked at run time and a variable can hold values of different types.

E

e-commerce 2
Buying and selling goods and services over the internet.

e-mail 2
Electronic mail, messages sent over the internet.

early exit (loop) 13
Leaving a loop before it would normally end.

early return 18
Returning from a method before the end of its body, usually when a condition is met.

edge case / boundary case 11
An input at the limit of what a program accepts, which often reveals errors.

Editor 8
The Word pane that checks spelling, grammar and writing style.

effect 10
A visual or sound change applied to an object, such as a fade or fly-in.

efficiency 3
How well an algorithm or device uses time, memory or energy.

EFI system partition 6
A small disk partition that holds the files UEFI needs to boot the system.

element 19
One item stored in an array or list.

element (item) 17
One value stored in an array.

else-if chain 16
A series of if and else if tests, of which the first true branch is executed.

embed fonts 10
Saving the fonts used in a presentation within the file so it looks right on other computers.

embedded software 11
Software built into a device to control it, such as in a washing machine.

embedded system 1
A computer built into a larger device to control one particular function.

empty array 19
An array with zero elements.

empty statement 15
A statement that does nothing, written as a lone semicolon.

encoding 4
Representing data, such as text or numbers, as bits.

encryption 2
Converting data into an unreadable form so that only authorized parties can read it.

end of file (EOF) 6
The condition that shows there is no more data to read from a file.

end task 7
A Task Manager command that ends a program that is not responding.

end-user license agreement (EULA) 6
The license agreement that sets out the terms of use of software.

endianness (big / little endian) 4
The order in which the bytes of a multi-byte number are stored in memory.

endnote 8
A note placed at the end of a document that gives a source or a comment.

enhanced for loop (for-each) 19
A for loop that goes through every element of an array or collection without an index.

entrance / emphasis / exit 10
The three kinds of animation: how an object appears, is highlighted and leaves.

entry point 14
The place where a program starts running, such as the main method in Java.

environment variable (PATH, JAVA_HOME) 14
A named setting of the operating system, such as PATH, that programs can read.

epsilon (tolerance) 15
A very small tolerance used to compare floating-point numbers for equality.

error alert 9
A message shown when invalid data is entered in a cell with data validation.

error message 14
A text that the compiler or system displays when it finds a problem.

error message (Excel) 9
A message that Excel displays in a cell when a formula cannot be calculated, such as #DIV/0!.

error value 9
A code in a cell that shows a failed calculation, such as #N/A or #REF!.

error-correcting code (ECC) 1
A memory technology that detects and corrects bit errors.

escape sequence 14
A backslash followed by a character that stands for a special character, such as \n for a new line.

Euclid's algorithm 5
An algorithm that finds the greatest common divisor of two numbers by repeated division.

exact match 9
A lookup that finds only a value identical to the search key (VLOOKUP with FALSE).

exception 14
An event that occurs while a program runs and disrupts its normal flow.

executable 7
A file that the operating system can run directly.

executable file 11
A file that contains a program the operating system can run, such as .exe.

execute / run 11
To carry out a program's instructions.

execution 13
The act of carrying out a statement or program.

execution model 14
The way a program is run, such as through an interpreter or the JVM.

expansion card 3
A circuit board that adds capabilities, such as graphics, to a computer.

exponent 9
The power to which a number is raised.

exponent / base 17
In a power, the exponent is the number of times the base is multiplied by itself.

exponential (growth) 21
Growth that multiplies at each step, for example doubling.

export 8
To save data from a program into a file in another format.

expression 1
A combination of values, variables and operators that produces a value.

expression tree 15
A tree structure that shows the order in which the parts of an expression are evaluated.

extract all 7
The Windows command that unpacks all the files of a compressed folder.

F

factorial 12
The product of all positive integers up to n, written n!.

fairness 5
The property that a scheduler gives all processes a fair share of the CPU.

fall-through 16
The behavior in a switch in which execution continues into the next case when there is no break.

FCFS (first come, first served) 6
A scheduling policy that serves processes in the order of their arrival.

feedback 11
Information about the result of an action that is used to improve the next step.

fetch / decode / execute 4
The three steps the CPU repeats for every instruction.

fetch-decode-execute cycle 4
The repeating cycle in which the CPU fetches an instruction, decodes it and executes it.

Fibonacci sequence 17
A sequence in which each number is the sum of the previous two: 0, 1, 1, 2, 3, 5.

field 4
A named data member of a class or a part of a record.

FIFO (first in, first out) 6
A queue discipline in which the oldest item is removed first.

file 3
A named collection of data stored on a storage device.

file association 7
The link between a file type and the program that opens it.

file descriptor 6
A small number that identifies an open file for the operating system.

File Explorer 7
The Windows program for browsing and managing files and folders.

file extension 7
The characters after the dot in a file name that show its type, such as .docx.

file format 4
The way data is organized in a file, such as JPEG or PDF.

file name extension 7
The characters after the dot in a file name that show its type.

file system 6
The way an operating system organizes, names and stores files on a disk.

file type 7
The category of a file, such as a document or an image, shown by its extension.

fill (color) 9
The color or pattern that fills a cell or shape.

fill handle 9
The small square at the corner of a selected cell that is dragged to copy a formula or continue a series.

filter 9
A tool that shows only the rows that meet a condition.

filtering 9
Showing only the rows that meet certain criteria.

Find and Replace 8
A Word command that searches for text and optionally replaces it.

finiteness 12
The property that an algorithm ends after a finite number of steps.

firewall 2
A system that filters network traffic according to security rules.

firmware 3
Software stored in a hardware device that controls its basic functions.

first-line indent 8
The indent of only the first line of a paragraph.

fixed program 5
A program that is built into the hardware and cannot be changed, as in early computers.

flag (variable) 13
A variable, often boolean, that records whether a condition has occurred.

Flash Fill 9
An Excel feature that fills cells automatically by recognizing a pattern in your entries.

flash memory 3
Non-volatile memory that can be erased and rewritten, used in USB drives and SSDs.

floating point 4
A way of representing real numbers with a mantissa and an exponent.

floating-point / decimal number 15
A number with a fractional part, such as 3.14.

floating-point number 15
A number type with a fractional part, such as float and double in Java.

flow 13
The movement from one step to the next.

flow line / arrow 13
The arrow in a flowchart that connects symbols and shows the direction of flow.

flowchart 13
A diagram that shows the steps of an algorithm with symbols and arrows.

folder 7
A container used to organize files on a disk.

folder (directory) 6
A container used to organize files and other folders on a disk.

font 8
The design of the letters of a text, such as Arial or Times New Roman.

font size / point 8
The size of text, measured in points; one point is 1/72 inch.

footnote 8
A note at the bottom of a page that gives a source or a comment.

form factor 3
The size, shape and layout of a component, such as ATX for motherboards.

formal parameter 18
A parameter as declared in the method header.

format (a disk) 6
To prepare a disk for use by creating a file system on it.

Format Page Numbers 8
A Word command that sets how page numbers look and where numbering begins.

Format Painter 8
A Word tool that copies the formatting of one piece of text to another.

format specifier 15
A code in a format string, such as %d or %.2f, that describes how a value is printed.

format string 15
The string that describes how output should be laid out, as in printf.

formatted output 15
Output laid out with controlled widths, decimals and alignment, as with printf.

formatting 8
Changing the appearance of text, paragraphs or a document, such as font, size, color and spacing.

formatting (printf) 20
Controlling how values are laid out in output, for example with printf and format specifiers.

formatting marks 8
Non-printing symbols, such as paragraph marks and spaces, that Word can display.

formula 9
An expression in a cell that starts with = and calculates a value from other cells.

formula bar 9
The bar above the worksheet that shows and edits the contents of the active cell.

fragment 2
A piece of a file or message, such as a part of a split resource.

fragmented 7
Describes a disk on which the pieces of files are scattered, which slows access.

frame 1
One of the still images that make up a video.

frame (page frame) 6
A fixed-size block of physical memory into which a page is loaded.

frames per second (fps) 1
The number of video frames shown per second.

freeze panes 9
An Excel command that keeps chosen rows and columns visible while the rest of the sheet scrolls.

frequency 19
How many times a value occurs in a data set.

frequency band 2
A range of radio frequencies used by a wireless network, such as 2.4 GHz or 5 GHz.

full adder 4
A circuit that adds two bits and a carry-in and produces a sum and a carry-out.

function 11
A named, reusable block of code that may take inputs and return a result.

function (Excel) 9
A predefined formula in Excel, such as SUM or AVERAGE, that performs a calculation.

function call 18
A statement that runs a function and passes it arguments.

function keys 3
The keys F1 to F12, which perform special tasks depending on the program.

functional programming 11
A style of programming that builds programs from functions without changing state.

functionality 6
What a system can do; the set of features it provides.

G

game controller 3
A device, such as a gamepad, used to control a game.

Gantt chart 6
A bar chart that shows the schedule of tasks over time.

garbage collection 14
The automatic freeing of memory that a program no longer uses.

garbage collector 20
The part of the runtime that finds and frees memory that is no longer used.

general purpose 5
Designed to be used for many kinds of tasks, not one particular task.

general-purpose language 11
A programming language made for many kinds of applications, such as Java or Python.

GPL (GNU General Public License) 6
A free-software license that requires derived works to be released under the same terms.

GPT (GUID Partition Table) 6
A disk partitioning scheme that supports very large disks and many partitions.

graphical user interface (GUI) 6
An interface in which the user works with windows, icons and menus rather than typing commands.

graphics card 3
A card that produces the image for a monitor and may perform graphics calculations.

graphics processing unit (GPU) 3
A processor specialized in graphics and highly parallel calculations.

greatest common divisor (GCD) 18
The largest integer that divides two integers without a remainder.

grid / cell 17
A pattern of rows and columns in which each box is a cell.

gridlines / guides 10
Lines and guides in PowerPoint that help to align objects.

group 10
To combine several objects so that they can be moved and formatted as one.

guest operating system 6
The operating system that runs inside a virtual machine.

H

half adder 4
A circuit that adds two bits and produces a sum and a carry.

halting problem 5
The problem of deciding whether a program will stop for a given input, which no algorithm can solve in general.

hand tracing (desk check) 12
Checking an algorithm by following it by hand with sample data.

handle (Windows) 6
A value that a program uses to refer to a system resource such as a file or a window.

handouts 10
Printed pages of slides, often with space for notes, given to the audience.

hanging indent 8
An indent in which every line except the first is indented, as used in bibliographies.

haptic feedback 3
Feedback that the user feels through vibration or touch.

hard disk drive (HDD) 3
A storage device that stores data magnetically on spinning platters.

hardware 3
The physical parts of a computer that can be touched.

hardware security key 2
A physical device used as a second factor to sign in securely.

Harvard architecture 4
A design with separate memories and buses for instructions and data.

hash (cryptographic) 2
A fixed-length value computed from data that cannot be reversed and changes when the data changes.

HDMI / DisplayPort / VGA 3
Video connectors used to attach a monitor or projector to a computer.

header 2
The part of a message or packet that holds control information such as addresses.

header / footer 8
Text that is repeated at the top (header) or bottom (footer) of every page.

header row 8
The first row of a table, which holds the column titles.

heading 8
A title of a section in a document, formatted with a heading style.

heap 16
The area of memory in which objects are created and live until they are no longer referenced.

heap (memory) 19
The memory region where objects and arrays are allocated.

helper method 21
A small method written to do one part of the work of a larger method.

hexadecimal 4
Related to the base-16 system that uses the digits 0 to 9 and A to F.

hidden 7
A file attribute that keeps a file from showing in the normal listing.

hidden slide 10
A slide that is part of the file but skipped during the slide show.

high-level / low-level language 11
A high-level language is close to human language; a low-level language is close to the machine.

high-level language 4
A programming language that is independent of the hardware and close to human language.

histogram 9
A column chart that shows how often values fall into intervals.

hold and wait 6
A condition for deadlock in which a process holds some resources while waiting for others.

hop 2
The passage of a packet through one router on its way to the destination.

host operating system 6
The operating system installed directly on the physical computer.

HTML / CSS / JavaScript 2
The three languages of web pages: structure, appearance and behavior.

HTTP / HTTPS 2
The protocols for transferring web pages; HTTPS is the encrypted version.

hybrid 11
Made of a mixture of two approaches.

hybrid kernel 6
A kernel that combines features of monolithic and microkernel designs.

hyperlink 8
Text or an object that opens another location, such as a web page, when clicked.

hypervisor 6
Software that creates and runs virtual machines.

hyphenation 8
Splitting words at the end of a line with a hyphen.

I

I/O (input/output) 6
Input/output; the transfer of data into and out of a computer.

IaaS / PaaS / SaaS 2
The three cloud service models: infrastructure, platform and software as a service.

icon 7
A small picture that stands for a file, folder or program.

icon sets 9
A conditional format that shows an icon in a cell according to its value.

IDE (integrated development environment) 11
An integrated development environment, a program that combines an editor, compiler and debugger.

identifier 14
A name given to a variable, method, class or other item in a program.

identity element 17
A value that leaves other values unchanged under an operation: 0 for addition, 1 for multiplication.

IEEE 754 4
The standard for representing floating-point numbers in binary.

if-else statement 16
A statement that runs one block if a condition is true and another if it is false.

immutability 20
The property that an object cannot be changed after it is created.

immutable 20
Unable to be changed after creation, like a String in Java.

imperative knowledge 5
Knowledge of how to do something, expressed as a sequence of steps.

imperative programming 11
A style of programming in which the program is a sequence of commands that change state.

implementation 11
Turning a design into working code.

implicit conversion 15
Automatic conversion by the compiler from one type to another without a cast.

in line with text 8
A Word wrapping style that places a picture on a line of text as if it were a character.

in-place 19
Done within the same array or memory, without a second structure.

increment / decrement 15
Operators that raise (++) or lower (--) a value by one.

indent 8
The space between the margin and the beginning of a paragraph or line.

indentation 12
Moving lines of code to the right to show their structure and make them easier to read.

index (array) 19
The position of an element in an array, starting at 0.

index (search engine) 2
The structure that a search engine builds from web pages so that it can find them quickly.

index (Word) 8
An alphabetical list of terms in a document with the pages they appear on.

indexing 20
Accessing an element by its position, such as with charAt or array[i].

infinite loop 17
A loop whose condition never becomes false, so it never ends.

infinite recursion 21
A recursion that has no base case reached and never stops, usually ending in a stack overflow.

information 1
Data that has been processed and given meaning.

information security (CIA) 2
Protecting confidentiality, integrity and availability of information.

information technology (IT) 1
The use of computers and networks to store, process and send information.

initial value 13
The value a variable has at the start.

initialization (initial value) 15
Giving a variable its first value when it is declared.

initialization (loop) 17
The statement at the start of a loop that sets the loop variable's first value.

inkjet / laser printer 3
Two printer types: inkjet printers spray ink, laser printers use toner and a laser.

inner loop 13
A loop that is placed inside another loop.

inner loop / outer loop 17
In nested loops, the inner loop runs completely for each step of the outer loop.

input (data) 11
The data given to a program or algorithm to work on.

input device / input unit 3
A device or unit that sends data into the computer, such as a keyboard.

input message 9
A message that appears when a cell with data validation is selected.

input size 12
The amount of data that an algorithm processes, usually written n.

input unit 3
The part of a computer that receives data from outside.

input validation 17
Checking that the data typed by the user is acceptable before using it.

input/output symbol 13
The parallelogram symbol in a flowchart that shows input or output.

insertion sort 12
A sorting algorithm that inserts each element into its correct place in the sorted part.

install / uninstall 7
To add or remove a program from a computer.

instant messaging 2
Real-time text communication between people over the internet.

instruction 4
One operation that the processor can perform, such as ADD or MOV.

instruction / command 14
A statement that tells the computer to do something.

instruction format 4
The layout of the fields of an instruction, such as the opcode and operands.

instruction register (IR) 4
The CPU register that holds the instruction currently being executed.

instruction set 3
The set of all instructions that a processor can execute.

integer 4
A whole number without a fractional part, such as -3, 0 or 42.

integer division 15
Division of two integers that gives an integer result, dropping the remainder.

integrated / discrete GPU 3
A GPU built into the processor chip, versus a separate, more powerful graphics card.

integrated development environment (IDE) 6
A program that combines editor, compiler and debugger for writing software.

integrity 2
The property that data is accurate and has not been changed without permission.

interface 3
A boundary through which two parts of a system communicate, such as a port or a user interface.

interference 2
Unwanted signals that disturb wireless communication.

intermediate code 11
Code between source code and machine code, such as bytecode.

Internet 2
The worldwide network of networks that connects computers.

Internet of Things (IoT) 1
A network of everyday devices with sensors and software that exchange data.

Internet service provider (ISP) 2
A company that provides access to the internet.

interpret 11
To execute a program line by line without translating it all first.

interpreted 11
Describes a language whose programs are run by an interpreter.

interpreter 11
A program that translates and runs source code statement by statement.

interrupt 6
A signal that makes the processor stop its current work to handle an event.

invalid input 12
Data that does not meet the program's requirements.

inverted index 2
A data structure that maps each word to the documents that contain it.

IP (Internet Protocol) 2
The protocol that addresses and routes packets across networks.

IP address 2
A number that identifies a device on a network.

IPv4 / IPv6 2
The two versions of the IP protocol, with 32-bit and 128-bit addresses.

iteration 17
One repetition of a loop body.

iteration (one pass of a loop) 12
One execution of the body of a loop.

iteration (one pass) 17
One execution of the body of a loop.

iterative 11
Using loops rather than recursion to repeat work.

iterative (version) 21
A solution that uses a loop instead of recursion.

J

jagged array 19
A two-dimensional array whose rows can have different lengths.

Java Development Kit (JDK) 14
The set of tools for developing Java programs, including the compiler.

Java Runtime Environment (JRE) 14
The software needed to run Java programs, which includes the JVM and libraries.

Java Virtual Machine (JVM) 14
The virtual machine that runs Java bytecode on any platform.

Javadoc / API documentation 14
The tool that creates HTML documentation from comments in Java code.

JIT (just-in-time) compiler 11
A compiler that translates bytecode into machine code while the program runs.

journaling 6
Recording changes in a log before they are made so a file system can recover after a crash.

just-in-time compilation (JIT) 14
Translating bytecode to machine code at run time to speed up execution.

justify 8
Aligning text so that both left and right edges are straight.

K

kernel 6
The core of an operating system that manages the CPU, memory and devices.

kernel (OS) 6
The central part of the operating system with full access to the hardware.

kernel mode 6
The privileged mode in which the operating system code runs.

key (insertion sort) 12
The element that insertion sort is currently placing into the sorted part.

key agreement (key exchange) 2
A method for two parties to agree on a shared secret key over an insecure channel.

keyboard 3
An input device with keys for typing characters.

keyboard layout 7
The arrangement of characters on the keys, such as QWERTY.

keyboard shortcut 7
A combination of keys that performs a command, such as Ctrl+C.

keyword 14
A word with a special meaning in a language that cannot be used as a name, such as class.

kibibyte / mebibyte / gibibyte 1
Binary units of size based on powers of 1024: kibibyte, mebibyte and gibibyte.

kilobyte / megabyte / gigabyte / terabyte 1
Units of data size: a kilobyte is 1,000 bytes and a megabyte 1,000 kilobytes, and so on.

kiosk (browsed at a kiosk) 10
A presentation mode in which a show runs by itself at a public information point.

L

labeled break 17
A break statement with a label that ends an outer loop instead of the inner one.

lane (PCIe) 3
One path of a PCI Express link, made of a pair of wires for each direction.

laser pointer 10
A pointer that shows a spot on the screen with a laser light.

latency 3
The delay before a transfer or response begins.

layer 2
One level of a layered system, such as a network protocol layer.

least common multiple (LCM) 18
The smallest positive number that is a multiple of two or more numbers.

length 19
The number of elements in an array or characters in a string.

level 8
A rank of an item in a hierarchy, such as a list level or heading level.

lexical analysis 11
The compiler phase that splits the source code into tokens.

lexicographic order 20
An ordering based on the character codes, as in a dictionary.

library 6
A collection of ready-made code or system files that programs can use.

license 6
The terms under which software may be used.

LIFO (last in, first out) 21
A stack discipline in which the item added last is removed first.

line (of code/output) 13
One line of source code or of program output.

line chart 10
A chart that connects data points with lines to show change over time.

line number 14
The number of a line in a source file, shown in error messages.

line spacing 8
The space between lines of text in a paragraph.

linear recursion 21
A recursion in which each call makes at most one further call.

linear search 12
A search that checks the elements one by one until the target is found.

linked / embedded object 10
A linked object is stored in another file and updated from it; an embedded object is stored inside the document.

linker 11
A program that joins object files and libraries into an executable.

linking error 11
An error that occurs when the linker cannot find a needed piece of code.

Linux distribution 6
A version of Linux packaged with software and tools, such as Ubuntu.

liquid crystal 3
A material between solid and liquid that is used in LCD screens.

list 19
An ordered collection of items that can grow or shrink.

literal 14
A fixed value written directly in code, such as 10 or "text".

literal / constant 20
A fixed value written directly in code.

loader 11
A program that loads executable code into memory to be run.

local account 7
An account stored only on the computer, not tied to an online service.

local area network (LAN) 2
A network that covers a small area, such as a building.

local variable 15
A variable declared inside a method or block and visible only there.

locale 15
The regional settings, such as language and number format, that affect output.

lock (screen) 7
To protect the screen with a password, using Win+L in Windows.

log (journal) 6
A record of events kept in a file.

logging 6
Recording events to a log for later review.

logic error 14
An error where the program runs but produces wrong results.

logic gate 4
A circuit that performs a basic logic operation, such as AND, OR or NOT.

logical address 6
The address that a program uses, which the system translates to a physical address.

logical operation 1
An operation on truth values, such as AND, OR and NOT.

logical operator 16
An operator such as &&, || and ! that combines boolean values.

logical test 9
A condition in a function such as IF that evaluates to TRUE or FALSE.

logical value (TRUE/FALSE) 9
A value that is either TRUE or FALSE.

long-term support (LTS) 14
A version of software that receives updates for an extended period, such as Java 21.

longest prefix match 2
The routing rule that selects the matching route with the most specific network prefix.

loop 13
A structure that repeats a block of statements while a condition holds.

loop / cycle 1
A sequence of steps that repeats.

loop body 13
The statements that a loop repeats.

loop invariant 12
A condition that is true before and after every iteration of a loop.

loop variable 17
The variable, such as i, that controls how many times a loop runs.

lossless compression 1
Compression after which the original data can be restored exactly.

lossy compression 1
Compression that discards some data, so the original cannot be restored exactly.

low-level 3
Close to the hardware, with little abstraction.

LRU (least recently used) 6
A replacement policy that removes the item that has not been used for the longest time.

M

MAC address 2
A hardware address that identifies a network card.

machine code 11
Instructions in binary form that the processor can execute directly.

machine instruction 4
One instruction in the binary form that the processor executes.

machine language 11
The language of the processor, made of binary instructions.

macro 8
A recorded sequence of commands that can be run again to automate a task.

magic number 11
A number in code with an unexplained meaning, which should be replaced by a named constant.

magnifier 7
A Windows tool that enlarges part of the screen.

mail merge 8
A Word feature that produces personalized copies of a document from a data source.

mail server 2
A server that sends, receives and stores e-mail.

mailbox 2
The place on a mail server where a user's messages are stored.

main document 8
The document in a mail merge that holds the fixed text and merge fields.

main memory 3
The memory, usually RAM, that holds the programs and data currently in use.

main method 14
The method with which a Java program starts running.

mainframe 1
A large, powerful computer used by organizations for critical, high-volume processing.

maintainability 6
How easily software can be corrected and updated.

maintenance 11
Changing software after delivery to fix errors or improve it.

malware 2
Software designed to harm a computer or steal data.

margin 8
The blank space between the text and the edge of the page.

markup 8
Special tags added to text to describe its structure or formatting.

markup language 11
A language that annotates text with tags, such as HTML.

matrix 19
A two-dimensional array organized in rows and columns.

MBR (Master Boot Record) 6
The first sector of a disk, with the partition table and boot code in older systems.

median 19
The middle value of a sorted set of numbers.

memoization 21
Storing the results of function calls to avoid computing them again.

memory 4
The part of a computer that stores programs and data.

memory address 4
A number that identifies a location in memory.

memory allocation 6
The reserving of memory for a program or an object.

memory cell 4
The smallest addressable unit of memory.

memory hierarchy 3
The arrangement of memory types from fast and small to slow and large.

memory management 6
The operating system task of allocating, tracking and freeing memory.

memory management unit (MMU) 6
A hardware unit that translates virtual addresses into physical addresses.

memory protection 6
Preventing a process from accessing memory that belongs to another process.

menu 17
A list of options that the user can choose from.

merge (version control) 11
To combine the changes of two branches into one.

merge cells 8
To combine several cells into a single larger cell.

merge conflict 11
A clash between changes to the same part of a file in different branches.

merge field 8
A placeholder in the main document that is replaced by data from the source.

merge sort 12
A sorting algorithm that splits the list, sorts each half and merges them.

message 2
A piece of information sent from one party to another.

metadata 6
Data that describes other data, such as a file's size or author.

method 18
A named block of code in a class that performs a task.

method call 18
A statement that runs a method.

method overloading 18
Defining several methods with the same name but different parameters.

method signature 18
A method's name together with its parameter types.

metropolitan area network (MAN) 2
A network that covers a city or a large campus.

microcontroller 1
A small computer on a single chip with processor, memory and input/output.

microkernel 6
A kernel with only essential functions, with other services running in user space.

middleware 6
Software that sits between the operating system and applications to connect them.

minimize / maximize / restore down / close 7
The four window controls that shrink, enlarge, return to the previous size or close a window.

mixed reference 9
A reference in which only the row or only the column is fixed, such as $A1.

mobile device 1
A portable device such as a phone or tablet.

modem 2
A device that converts digital data to signals for a line and back.

modified Harvard architecture 4
A Harvard design in which instructions can also be treated as data.

modulo (mod) 5
The operation that gives the remainder of a division.

monitor / screen 3
The output device that displays images and text.

monolithic kernel 6
A kernel in which all operating system services run in one large program.

Morph 10
A PowerPoint transition that smoothly changes one slide into another.

motherboard 3
The main circuit board that connects the parts of a computer.

motion path 10
A line along which an object moves during an animation.

mouse 3
An input device that moves a pointer on the screen.

move 7
To change the location of a file, folder or object.

multi-factor authentication 7
A sign-in method that needs two or more kinds of proof, such as a password and a code.

multi-factor authentication (MFA) 2
A login method that requires two or more independent proofs of identity.

multi-level sort 9
A sort by several columns, one after the other.

multi-way decision 13
A decision with more than two possible outcomes.

multidimensional array 19
An array with two or more dimensions.

multifunction device 3
A device that combines functions, such as printing, scanning and copying.

multilevel list 8
A list with items at several levels of indentation.

multiprogramming 6
Keeping several programs in memory so the CPU is always busy.

mutable 20
Able to be changed after creation.

mutual exclusion 6
The rule that only one process may use a shared resource at a time.

N

name box 9
The box left of the formula bar that shows the address or name of the selected cell.

named range 9
A group of cells that has been given a name.

naming convention 15
The agreed style for choosing names, such as camelCase for variables.

NaN (Not a Number) 15
The special floating-point value for an undefined result, such as 0.0/0.0.

narrowing conversion 15
Converting a value to a smaller type, which can lose data and needs a cast.

NAT (network address translation) 2
A technique that lets many devices share one public IP address.

navigation pane 8
The Word pane that lists headings and helps you move around a document.

negation 9
The reversal of a condition's truth value, for example with NOT.

nested 13
Placed inside another of the same kind.

nested decision 16
A decision placed inside another decision.

nested loop 13
A loop placed inside another loop.

network 3
A group of connected computers and devices.

network / cloud storage 3
Storage that is reached over a network or over the internet.

network byte order 4
The byte order used for data on the network, which is big-endian.

network interface card 3
A card that connects a computer to a network.

network interface card (NIC) 3
The hardware component that connects a computer to a network.

neural processing unit (NPU) 3
A processor specialized for AI calculations.

newline 15
The character that ends a line and begins the next.

newline / line terminator 14
The character or characters that mark the end of a line.

no preemption 6
The condition that a resource cannot be taken from a process by force.

non-breaking space 8
A space that keeps two words together on the same line.

non-volatile / persistent 3
Memory that keeps its contents without power.

not responding 7
The state of a program that no longer reacts to input.

notification center 7
The Windows area that collects notifications and quick actions.

null reference 19
A reference that points to no object.

number format 9
The way a number is displayed in a cell, such as currency or percentage.

numbered list 8
A list in which items are numbered in order.

O

object 14
A unit that combines data and behavior; an instance of a class.

object file 11
The file with machine code that a compiler produces before linking.

object-oriented 11
Describes a language or approach that is built around objects and classes.

octal 4
Related to the base-8 system that uses the digits 0 to 7.

octet 2
A group of eight bits, as used in networking.

off-by-one error 17
A mistake in which a loop or index runs one time too many or too few.

off-page connector 13
A flowchart symbol, a pentagon, that continues the flow on another page.

offset 6
The distance from a starting address or position.

on-premises 2
Hosted on the organization's own premises rather than in the cloud.

one's complement 4
Connected to a network, especially the internet.

online 2
Connected to the internet or available over a network.

opcode 4
The part of a machine instruction that says which operation to perform.

open source 6
Software whose source code is available to everyone to use and modify.

operand 15
A value on which an operator acts.

operand stack 14
The stack that the JVM uses to hold operands while executing bytecode.

operating system 1
The software that manages the hardware and runs other programs.

operating system (OS) 6
The software that manages the hardware and lets programs run.

operator 15
A symbol that stands for an operation, such as + or *.

operator (Excel) 9
A symbol in a formula that specifies a calculation, such as +, & or >.

operator precedence 9
The rules that decide which operator is evaluated first.

optical disc / external disk 3
Storage media such as DVDs or portable drives.

optimize (drives) 7
To rearrange or trim a drive so that it works more efficiently.

orientation 8
The direction of a page, portrait or landscape.

out of bounds (index) 19
An index that lies outside the valid range of an array.

outer loop 13
The loop that contains an inner loop.

outline 10
A view of a presentation that shows only the text of the titles and body.

output (data) 11
The data that a program produces.

output device / output unit 3
A device or unit that presents results, such as a monitor or printer.

output stream 14
The flow of data from a program to a destination such as the screen.

output unit 3
The part of a computer that presents results to the user.

overflow 4
A result that is too large for the number of bits available.

overflow (integer) 21
A result that is too large to fit in an int, so it wraps around.

P

package (Java) 14
A group of related classes, with a name that avoids naming conflicts.

package for CD (PowerPoint) 10
A PowerPoint feature that bundles a presentation with its linked files.

package manager 11
A tool that installs and updates software packages.

packet 2
A small unit of data sent across a network.

packet switching 2
A method of sending data in small packets that travel independently.

page 6
A fixed-size block of memory or storage used in paging.

page break 8
A mark that forces the following text onto a new page.

page fault 6
An event that occurs when a program uses a page that is not in memory.

page file 6
A file on disk used as an extension of memory.

page number 8
A number placed on each page, usually in the header or footer.

page replacement 6
Choosing which page to remove from memory when a new page must be loaded.

page table 6
The table that maps virtual pages to physical frames.

paging 6
A memory management technique that divides memory into fixed-size pages.

palindrome 12
A word or sequence that reads the same forwards and backwards.

paragraph 8
A block of text that ends with the Enter key.

paragraph spacing 8
The space before and after a paragraph.

parallelism 4
Doing several operations at the same time.

parameter 18
A variable in a method declaration that receives a value when the method is called.

parent directory 7
The folder that contains the current folder.

parity bit 1
An extra bit that is added to make the number of ones even or odd, to detect errors.

parsing / syntax analysis 11
The compiler phase that checks the structure of the code against the grammar.

partition 6
A separate section of a disk that is treated as an independent drive.

partition / volume 7
A section of a disk that is formatted and assigned a drive letter.

partitioning scheme 6
The way a disk is divided into partitions, such as MBR or GPT.

pass (sorting) 12
One complete walk through the data in a sorting algorithm.

pass by value 18
A method of passing arguments in which the method receives a copy of the value.

passkey 2
A passwordless credential that is based on a cryptographic key pair.

password 2
A secret word or phrase used to prove identity.

password manager 2
A program that stores and fills in a user's passwords securely.

path 6
The route that leads to a file or folder, written as a list of folders.

pattern (coding) 17
A recurring structure that a solution follows in code.

pattern (output) 17
A shape built from characters printed with loops, such as a triangle of stars.

pattern recognition 5
Finding similarities or regularities in problems or data.

perfect number 17
A number equal to the sum of its proper divisors, such as 6.

performance 3
How fast and how well a system does its work.

peripheral 6
A device connected to a computer, such as a printer or keyboard.

permanent delete 7
Deleting a file so that it does not go to the Recycle Bin.

permission 6
The right to read, write or run a file or resource.

personal area network (PAN) 2
A very small network around a person, such as Bluetooth devices.

personal computer 1
A computer designed for use by one person.

personal data 2
Information that can be used to identify a person.

phishing 2
A fraud in which a fake message tricks a user into revealing private information.

physical address 6
The real address of a location in RAM.

picture / image 8
A graphic added to a document, such as a photo.

pie chart 10
A circular chart divided into slices that show shares of a whole.

PIN 7
A short number code used to unlock a device or account.

PivotTable 9
An Excel tool that summarizes a large data table by grouping and totaling.

pixel 1
The smallest element of a digital image.

pixel density (PPI) 3
The number of pixels per inch of a screen, which tells how sharp the image is.

placeholder 10
A box on a slide layout that is reserved for content such as text or a picture.

plain text 14
Text without any formatting.

plaintext 2
Data in readable form before it is encrypted.

platform 14
The system, such as an operating system and hardware, on which programs run.

platter 3
A circular disk in a hard disk drive on which data is stored.

plug and play 6
The ability of a system to detect and configure new hardware automatically.

pointer (mouse / laser) 10
A symbol that shows the position of the mouse or laser on a slide.

pointer (two-pointer technique) 19
An index that marks a position in an array in algorithms that use two positions.

pointing device 3
An input device that moves a cursor, such as a mouse.

polling rate 3
How many times per second a device such as a mouse reports its state.

port 3
A connection point on a device or a number that identifies a network service.

portable 11
Able to run on different platforms without changes.

portrait / landscape 8
The two orientations of a page: portrait (tall) and landscape (wide).

positional number system 4
A number system in which the value of a digit depends on its position.

POST (power-on self-test) 6
The check of the hardware that the computer runs when it is turned on.

post-test loop (do-while) 13
A loop that executes the body once before testing the condition (do-while).

postfix / suffix 15
A part added at the end of a word or name.

power supply unit (PSU) 3
The component that supplies power to the parts of a computer.

power-on self-test (POST) 6
A check of the hardware that the firmware performs at startup.

pre-test loop (while) 13
A loop that tests the condition before the body (while).

precision 4
The number of digits that a value can represent exactly.

predefined process 13
A flowchart symbol for a process defined elsewhere, such as a method.

prefix 15
A part added at the start of a word or name.

preparation (flowchart symbol) 13
A flowchart symbol, a hexagon, used for steps such as initialization.

presentation (act of presenting) 10
The act of showing a talk with slides to an audience.

presentation (file) 10
A file that contains slides, as in a PowerPoint presentation.

presenter 10
The person who gives a presentation.

Presenter View 10
A PowerPoint view for the speaker that shows notes and the next slide.

preview 8
A display of how a document will look before it is printed or applied.

prime number 12
A natural number greater than 1 with no divisors other than 1 and itself.

primitive type 15
A basic data type built into the language, such as int or boolean.

principle of least privilege 6
The rule that every program or user should have only the rights needed.

print area 9
The part of a worksheet that is selected for printing.

print preview 8
A view that shows how a document will look when printed.

printer 3
An output device that puts text or images on paper.

privacy 1
The right to keep personal information under one's own control.

private (local) IP address 2
An IP address used inside a local network that is not reachable from the internet.

private key 2
The secret key that its owner keeps to decrypt data or create signatures.

privileged 6
Having the extra rights needed to run sensitive operations.

problem definition 12
The statement of what a problem is, with its inputs and expected outputs.

problem solving 12
The process of finding a way to reach a goal from a given situation.

process 6
A program in execution.

process ID (PID) 6
The number that the operating system uses to identify a process.

process state 6
The state of a process, such as ready, running or waiting.

process symbol 13
The rectangle symbol in a flowchart that shows a calculation or action.

processing (IPOS stage) 1
The stage of the input-process-output-storage cycle in which data is transformed.

processor 3
The component that executes instructions; the CPU.

processor (CPU) 6
The central processing unit that executes program instructions.

program 6
A set of instructions that a computer executes.

program counter (PC) 4
A register that holds the address of the next instruction.

programming 11
Writing instructions that a computer can execute to solve a problem.

programming language 11
A formal language for writing programs, such as Java or Python.

programming paradigm 11
A style of programming, such as imperative, functional or object-oriented.

projector 3
A device that shows the computer's screen image on a wall or screen.

prompt 15
A text that asks the user for input, such as the Windows prompt or a message before reading a value.

properties 7
The information about a file or item, such as size, type and date.

protocol 3
A set of rules that two devices follow to communicate.

prototyping 11
Building a simple early version of a system to test ideas.

pseudocode 12
A way of writing an algorithm in plain, structured language that is not tied to any programming language.

public IP address 2
An IP address that is reachable from the internet.

public key 2
The key that is shared openly and used to encrypt data or verify signatures.

pure method/function 18
A method whose result depends only on its arguments and which has no side effects.

Q

quantization 1
Rounding sample values to a limited set of levels when digitizing a signal.

quantum computer 1
A computer that uses quantum mechanics, with qubits, to solve some problems faster.

qubit 1
The basic unit of quantum information, which can be 0, 1 or a combination of both.

query 1
A request for data from a database or search engine.

quick settings 7
A Windows panel with toggles for Wi-Fi, volume, brightness and other settings.

quotient 15
The result of a division.

R

RAM 6
Random access memory, the fast, volatile main memory of a computer.

random 12
Unpredictable; produced by chance.

random access 12
The ability to read any item directly, without reading the ones before it.

random access memory (RAM) 3
The fast, volatile memory in which running programs and their data are kept.

random number 17
A number produced by chance, for example with Math.random or the Random class.

range (Excel) 9
A group of adjacent cells, written as A1:B5.

range (wireless) 2
The distance over which a wireless signal can be received.

ransomware 2
Malware that locks a user's files and demands payment to unlock them.

read-only 7
A file attribute that allows reading but not changing.

read-only memory (ROM) 3
Read-only memory, which keeps its contents without power and holds firmware.

read-write head 3
The part of a disk drive that reads and writes data on the platter.

read/write head 5
The part of a disk drive, or of a Turing machine, that reads and writes symbols.

readability 13
How easily code or a diagram can be read and understood.

reading order 10
The order in which a screen reader reads the objects on a slide.

ready queue 6
The queue of processes that are ready to run and waiting for the CPU.

real-time protection 7
Antivirus protection that scans files and activity continuously.

real-time system 6
A system that must respond within strict time limits.

recipient 8
A person who receives a message or a mail-merge document.

record 4
A group of related fields treated as a unit, such as one row of a table.

record (DNS record) 2
An entry in the DNS that links a name to data such as an IP address.

recovery 7
Restoring a system or files after a failure or loss.

recovery key 7
A code that lets a user unlock an encrypted drive or an account.

recursion 21
A technique in which a method calls itself to solve smaller versions of a problem.

recursion depth 21
The number of nested calls that are active at the same time.

recursion tree / call tree 21
A diagram that shows all the calls made by a recursive method.

recursive case (step) 21
The part of a recursive method that makes the recursive call on a smaller problem.

Recycle Bin 7
The Windows folder that keeps deleted files until they are removed permanently.

reference 15
A value that refers to an object stored elsewhere in memory.

refresh rate 3
The number of times per second that a display redraws the image.

register 4
A small, very fast storage location inside the CPU.

regular expression 20
A pattern that describes a set of strings and is used for searching text.

rehearse 10
To practice a presentation before giving it.

Rehearse Timings 10
A PowerPoint feature that records how long you spend on each slide.

relative path 7
A path that starts from the current folder instead of the root.

relative reference 9
A cell reference that changes when the formula is copied, such as A1.

reliability 6
The ability of a system to work correctly over time without failing.

remainder 5
What is left over after a division.

remainder (modulo) 12
The remainder of an integer division, calculated with the % operator.

rename 7
To change the name of a file or folder.

repetition 13
The repeating of steps in an algorithm.

repetition (iteration) structure 16
A structure that repeats statements, such as for, while or do-while.

repetitive flow (iteration) 13
A flow in which steps are executed again while a condition holds.

Replace Fonts 10
A PowerPoint command that replaces a font used in a presentation with another.

request 2
A message in which a client asks a server for something.

request / response 1
The pattern in which a client sends a request and the server sends back a response.

requirement 11
A statement of what a system must do or achieve.

reserved name 7
A name that Windows does not allow for files, such as CON or NUL.

reset 7
To return a device or setting to its original state.

resolution 3
The number of pixels that a screen or image has, such as 1920 by 1080.

resource 6
Anything a program can use, such as memory, CPU time or a file.

response 2
A message that a server sends back to answer a request.

response time 3
The time a display or device needs to react to a change.

restore 7
To bring back a file or system from a backup.

restore point 7
A saved state of the system to which Windows can be returned.

return 18
To send a value back from a method to its caller.

return (a value) 18
To send a result back to the caller of a method.

return (value) 18
To send a value back from a method to the code that called it.

return type 18
The type of the value that a method returns, or void if it returns nothing.

return value 18
The value that a method sends back to its caller.

reuse 18
Using existing code, such as methods, again instead of rewriting it.

reversing (an array/string) 19
Changing the order of the elements of an array or the characters of a string to the opposite.

RGB (color model) 1
A color model that mixes red, green and blue light.

ribbon 8
The strip of tabs and commands at the top of Office applications.

ripple-carry adder 4
An adder built from full adders in which each carry passes to the next stage.

RISC / CISC 4
Two processor design styles: reduced and complex instruction set computers.

ROM / flash memory 6
Types of memory that keep their contents without power.

root (directory) 6
The top-level folder from which all other folders branch.

root (folder) 7
The top-level folder of a drive.

root server 2
One of the DNS servers at the top of the DNS hierarchy.

rotational latency 3
The time that a disk needs to rotate to the desired sector.

round robin 6
A scheduling method that gives each process a time slice in turn.

rounding 4
Replacing a number with a nearby number that has fewer digits.

router 3
A device that forwards packets between networks.

routing table 2
The table that a router uses to decide where to send packets.

row 8
A horizontal line of cells in a table or worksheet.

ruler 8
The scale at the top and left of a document window that shows margins and indents.

run 13
To start a program or to execute an algorithm.

run (run-length encoding) 1
A sequence of equal consecutive values in run-length encoding.

run-length encoding (RLE) 1
A compression method that replaces repeated values with a value and a count.

runtime 6
The time during which a program is running, or the software environment that supports it.

runtime error 14
An error that occurs while the program is running, such as division by zero.

S

sampling / sample 1
Measuring a signal at regular times to convert it to digital form.

sampling rate 1
The number of samples taken per second when a signal is digitized.

sandbox 6
An isolated environment in which a program runs with limited rights.

save / save as 8
Commands that store a document under its current name or under a new name.

scale 7
The relative size of an item compared to another, such as zoom or the size of a window.

scanner 3
A device that turns paper documents or images into digital files.

scatter chart 9
A chart that plots pairs of numbers as points on x and y axes.

scheduler 6
The operating system component that chooses which process runs next.

scheduling 6
Deciding the order in which processes use the CPU.

scope 15
The part of a program in which a name is visible.

screen 13
The display unit of the computer.

screen reader 7
Software that reads aloud the content of the screen for blind users.

screen resolution 7
The number of pixels of the screen, expressed as width by height.

screen size 3
The diagonal size of a display, measured in inches.

screenshot 7
An image of what is displayed on the screen.

scripting language 11
A language, such as Python or JavaScript, in which programs are interpreted from scripts.

scroll bar 7
A bar at the edge of a window used to scroll its content.

search 12
The process of finding an item in a collection.

search engine 2
A website, such as Google, that finds web pages for a query.

search operator 2
A special word or symbol, such as quotation marks or minus, that refines a web search.

search range (interval) 12
The part of the data in which a search algorithm still looks for the target.

section 8
A part of a document that can have its own page settings.

section break 8
A mark that ends a section and starts another with its own formatting.

sector 3
The smallest unit of storage on a disk track.

Secure Boot 6
A UEFI feature that allows only trusted software to start the system.

security 1
Protection of systems and data against threats.

seek time 3
The time it takes a disk head to move to the right track.

segment (TCP) 2
A unit of data in TCP, with a header and a part of the stream.

segmentation fault (access violation) 6
An error that occurs when a program tries to access memory that it is not allowed to use.

selection 8
The part of a document, table or sheet that is currently marked.

selection (conditional) 16
A structure that runs code only when a condition is true, such as if.

selection (decision) structure 16
A structure that chooses between paths by testing a condition.

selection pane 10
The PowerPoint pane that lists the objects on a slide so you can select or hide them.

selection sort 12
A sorting algorithm that repeatedly selects the smallest remaining element and places it first.

semantic analysis 11
The compiler phase that checks the meaning of the code, such as type correctness.

semantics 11
The meaning of the statements of a language, as opposed to their form.

semicolon 14
The punctuation mark ; that ends a statement in Java.

sensor 1
A device that measures a physical quantity, such as temperature or light.

sentinel value 17
A special value that marks the end of the input and stops a loop.

sentinel-controlled loop 13
A loop that continues until a special value is read.

separator 14
A character that divides items, such as a comma or a slash.

sequence (sequential structure) 16
A structure in which statements are executed one after another.

sequence number 2
A number that gives the order of a segment or packet in a stream.

sequence structure 16
A structure in which statements run one after another from top to bottom.

sequential flow (sequence) 13
A flow in which steps execute one after another in order.

sequential read 3
Reading data in the order in which it is stored.

server 1
A computer or program that provides services to clients.

session key 2
A temporary key used to encrypt the data of a single session.

settings 7
The Windows application with which settings are changed.

shadowing 18
The situation in which a local name hides another name of the same name in an outer scope.

shallow copy 19
A copy that duplicates the references to the objects but not the objects themselves.

shape 10
A drawn object, such as a rectangle or arrow, that is added to a slide.

sheet protection 9
A protection that prevents changes to the cells of a worksheet.

sheet tab 9
The tab at the bottom of the workbook that selects a worksheet.

short-circuit evaluation 15
Evaluation of a logical expression that stops as soon as the result is known.

shortcut (file, .lnk) 7
A small file that points to another file, folder or program.

Show Formulas 9
An Excel command that displays formulas instead of their results.

side effect 18
A change to something outside the method, such as a global variable or output.

Sieve of Eratosthenes 19
An algorithm that finds all prime numbers up to a limit by crossing out multiples.

sign / exponent / fraction (IEEE 754) 4
The three fields of an IEEE 754 floating-point number.

sign bit 15
The bit that shows whether a number is positive or negative.

sign in / sign out 7
To start or end a session by giving or withdrawing credentials.

signed 4
Describes a number type that can be positive or negative.

signed overflow 4
An overflow in signed arithmetic, when the result is out of range.

simultaneous 8
At the same time.

SJF (shortest job first) 6
A scheduling policy that runs the process with the shortest burst time first.

slide 10
One page of a presentation.

slide layout 10
The arrangement of placeholders on a slide.

slide master 10
The slide that controls the layout and formatting of all slides.

slide pane / notes pane 10
The areas of the PowerPoint window that show the slide and its notes.

slide show 10
The full-screen display of a presentation.

slide size 10
The dimensions of slides, such as widescreen 16:9.

slide sorter 10
A PowerPoint view that shows thumbnails of all slides so they can be reordered.

sliding window 12
A technique that moves a fixed-size range over data to process it efficiently.

slot 3
A socket on the motherboard into which a card or memory module is plugged.

SmartArt 10
A PowerPoint feature that turns text into diagrams and graphics.

SMTP / IMAP 2
Protocols for e-mail: SMTP sends messages and IMAP reads them on the server.

snap 7
The Windows feature that sizes windows to half or a corner of the screen.

Snipping Tool 7
A Windows tool that captures part of the screen as an image.

social engineering 2
Tricking people into revealing secrets or performing actions that help an attacker.

socket (CPU) 3
The place on the motherboard where the processor is installed.

software 1
The programs and data that tell the hardware what to do.

software development life cycle 11
The stages in building software, from requirements and design to testing and maintenance.

solid-state drive (SSD) 3
A storage device that uses flash memory and has no moving parts.

sort 9
To arrange rows or data in order according to a column.

sorted 12
Arranged in a defined order, such as ascending.

sorting 12
Arranging data in a defined order.

sound card 3
A card or chip that converts digital audio to sound.

source code 11
The program text written by humans in a programming language.

source file 14
A file that contains source code, such as one with the .java extension.

space complexity 12
The amount of memory that an algorithm needs as a function of input size.

spaghetti code 13
Code with tangled, hard-to-follow control flow, typically from unrestricted jumps.

spatial locality 3
The tendency to access memory locations that are near ones used recently.

speaker / headphones 3
Output devices that produce sound.

speaker notes 10
Notes written for the speaker that the audience does not see.

spelling and grammar 8
The Word feature that checks spelling and grammar.

spill 9
The behavior of a dynamic array formula whose result fills several neighboring cells.

spreadsheet 9
A program or document made of cells in rows and columns for calculations.

spyware 2
Malware that secretly collects information about a user.

square root 12
A number that, multiplied by itself, gives the original number.

SSD / HDD 7
Two storage types: the solid-state drive has no moving parts and the hard disk drive has spinning platters.

stack (memory) 18
The area of memory that stores the local variables and calls of methods.

stack frame 18
The block of the call stack that holds the parameters and local variables of one call.

stack machine 14
A virtual machine whose instructions work on an operand stack, like the JVM.

stack overflow 21
An error that occurs when the call stack becomes full, usually from endless recursion.

stack trace 14
A report of the method calls that were active when an error occurred.

standard account 6
An account with limited rights that cannot change system settings.

standard user 7
A user without administrator rights.

Start menu 7
The Windows menu that gives access to apps, settings and power options.

startup apps 7
Programs that start automatically when the user signs in.

state 5
The condition of a system at a given moment, such as the stored values of variables.

statement 14
A complete instruction in a program, usually ending with a semicolon.

static method 18
A method that belongs to the class itself and is called without an object.

static typing 11
A system in which types are known and checked at compile time.

status bar 7
The bar at the bottom of a window that shows information about the current state.

status code 2
A number in a response that shows the result of a request, such as 404.

step 12
One action in an algorithm or process.

storage 3
The devices and media that keep data when it is not being processed.

Storage Sense 7
A Windows feature that frees disk space by deleting unneeded files automatically.

stored program 4
The idea that the program is kept in memory along with the data.

string 4
A sequence of characters, such as text.

string concatenation 20
Joining strings together with the + operator or the concat method.

string literal 20
A fixed sequence of characters in double quotes in the code.

string pool 20
A memory area in which the JVM keeps unique string literals for reuse.

structured programming 13
A way of programming with sequence, selection and repetition and without arbitrary jumps.

structured reference 9
A reference in Excel that uses table and column names instead of cell addresses.

style 8
A named set of formatting settings applied to text in one step.

subdomain 2
A part of a domain name that precedes the main domain, such as mail in mail.example.com.

subfolder 6
A folder inside another folder.

subprocess 13
A process that is run by another process or a part of a process defined elsewhere.

subscription 6
An arrangement that gives access to software for a recurring fee.

subset 21
A set whose elements all belong to another set.

substring 20
A part of a string, taken with the substring method.

SUBTOTAL 9
An Excel function that calculates totals for the visible rows of a filtered list.

Summary / Section / Slide Zoom 10
Three kinds of PowerPoint zoom that link to a summary, a section or a slide.

supercomputer 1
The fastest and most powerful type of computer, used for heavy scientific calculations.

superscript / subscript 8
Characters set above or below the line, as in x2 and H2O.

sustainability 5
Meeting present needs without harming the future, including energy and resource use.

swap 12
To exchange the values of two variables or positions.

swap space 6
The area of disk that the operating system uses as an extension of memory.

swapping 6
Moving memory contents between RAM and disk.

swapping / swap space 6
Moving memory contents between RAM and the disk area reserved for it.

switch 2
A device or construct that selects between alternatives; in Java, a multi-way selection statement.

switch (network) 3
A device that connects computers in a network and forwards frames to the correct port.

switch expression 16
A switch that produces a value, written with arrows (case X -> value).

switch statement 16
A statement that selects one of several blocks according to the value of an expression.

symbol 5
A character or sign used to represent something.

symmetric encryption 2
Encryption in which the same key is used for encryption and decryption.

syntax 11
The rules that define how correct programs are written.

syntax error 14
An error found by the compiler because the code breaks the rules of the language.

syntax tree 11
A tree that shows the grammatical structure of the code.

system call 6
A request that a program makes to the operating system to perform a privileged operation.

system software 6
Software that manages the hardware and supports application software, such as an operating system.

system tray / notification area 7
The area at the right of the Windows taskbar that shows the clock and background programs.

System-on-Chip (SoC) 3
A chip that combines the CPU, GPU, memory controller and other parts, as in phones.

T

tab (character) 14
The character that moves the cursor to the next tab stop.

tab (ribbon / Explorer / tab stops) 8
A page-like element used to switch between pages, such as ribbon tabs or tab stops.

tab stop 14
A position on the ruler at which text aligns when the Tab key is pressed.

table 8
An arrangement of data in rows and columns.

table (Excel table) 9
A range of cells that is formatted and managed as a table with filtering and totals.

table of contents 8
A list of the headings of a document with page numbers.

table of figures 8
A list of the figures and captions in a document.

tail call optimization 21
An optimization that reuses the stack frame for the last call of a method.

tail recursion 21
A recursion in which the recursive call is the last operation in the method.

tape 5
The medium that stores symbols on a Turing machine.

target 12
The value or position that a search is looking for.

target audience 10
The group of people for whom a presentation is designed.

Task Manager 7
A Windows tool that shows running programs and lets the user end them.

task view 7
A Windows feature that shows open windows and virtual desktops.

taskbar 7
The bar at the bottom of the Windows screen that holds the Start button and open programs.

TCP / UDP 2
Two transport protocols: TCP is reliable and ordered, UDP is faster but without guarantees.

template 8
A file that holds the layout and formatting from which new documents are created.

temporal locality 3
The tendency to access again the same data that was used recently.

temporary file 7
A file that a program creates for short-term use.

temporary variable 12
A variable that holds a value for a short time, such as in a swap.

terminal (start/end) symbol 13
The rounded symbol that marks the start or end of a flowchart.

test 12
An activity that checks whether a program works correctly.

test case 12
A set of inputs and the expected output used to test a program.

testing 5
The process of running a program to find errors.

text 9
Characters in a cell that are not numbers or formulas.

text (string) 12
A sequence of characters.

text box 10
A box in which text can be typed and placed anywhere on a slide.

text editor 6
A program for creating and editing plain text, such as Notepad.

text file 11
A file that holds only characters, without formatting.

text wrapping 8
The way text wraps around a picture or object.

theme 8
A set of colors, fonts and effects that gives a document a consistent look.

thermal throttling 3
Slowing the processor down to prevent overheating.

thesaurus 8
A Word tool that suggests words with similar meanings.

thin client 1
A computer that depends on a server for most of its processing.

thread 3
A unit of execution within a process.

three-way handshake 2
The three messages that start a TCP connection: SYN, SYN-ACK and ACK.

throw 15
To signal an error with an exception or error object.

throw (an exception) 14
To signal an error by creating an exception object with the throw statement.

throw (exception) 21
To signal an error by raising an exception.

thumbnail 10
A small picture of a slide or file.

time complexity 12
The number of steps an algorithm takes as a function of its input size.

time complexity / Big O 21
A measure of how running time grows with the input size, expressed in Big O notation.

time sharing 6
A technique that lets many users share the CPU by giving each a short turn.

time slice (quantum) 6
The short period of CPU time that a process receives in round robin scheduling.

timer 10
A tool that counts time, such as one that controls a slide show.

timing 10
The time and order in which animations or slides appear.

title bar 7
The bar at the top of a window with the name of the document or program.

TLS 2
A protocol that encrypts communication between programs, used with HTTPS.

token 14
A unit of source code, such as a keyword or identifier, that the compiler identifies.

toolbar 7
A bar of buttons that give quick access to commands.

top-level domain (TLD) 2
The last part of a domain name, such as .com or .org.

total row 9
A row at the bottom of an Excel table that shows totals.

touch interface 6
An interface that is used with fingers on a touch-sensitive screen.

touchpad 3
A flat surface on a laptop that moves the pointer.

touchscreen 3
A display that responds to touch.

Towers of Hanoi 21
A classic puzzle in which disks are moved between pegs, often used to show recursion.

TPM 6
The Trusted Platform Module, a chip that stores encryption keys and checks system integrity.

trace (code tracing) 17
To follow the execution of code step by step by hand.

trace / tracing 12
Following the steps of an algorithm and recording the values.

trace table 12
A table that records the values of variables at each step of an algorithm.

traceroute 2
A command that shows the route that packets take to a destination.

track 3
A circular path on a disk platter.

track changes 8
A Word feature that marks all edits so they can be reviewed.

transfer (data) 3
The movement of data from one place to another.

transistor 1
A semiconductor switch that is the basic building block of digital circuits.

transition 10
The effect that is shown when one slide changes to the next.

transition rule 5
A rule of a Turing machine that says what to write, how to move and which state to enter.

translation 11
Converting a program from one language to another.

translator (compiler/interpreter) 11
A program that translates code from one language to another, such as a compiler or interpreter.

traversal (iterate over array) 19
Visiting each element of an array in turn.

trigger 10
An action, such as a click, that starts an animation.

TRIM 7
An Excel function that removes extra spaces from text.

Trojan horse 2
Malware that pretends to be useful software.

truncation 15
Cutting off the fractional part of a number without rounding.

truth table 4
A table that lists all input combinations of a logic function and its outputs.

TTL (time to live) 2
A number that limits the lifetime of a packet by the number of hops.

tunnel 2
An encrypted connection through a network, as used by a VPN.

Turing machine 5
An abstract model of computation made of a tape, a head and a set of rules.

turnaround time 6
The total time from the submission of a process to its completion.

two's complement 4
A way of representing negative numbers in binary by inverting the bits and adding one.

two-dimensional array 19
An array with rows and columns, accessed with two indices.

type 15
A kind of value, such as int or String, that determines what can be done with it.

type conversion 15
Changing a value from one type to another.

type error 14
An error caused by using a value of a wrong type.

type promotion 15
Automatic conversion of a smaller type to a larger one in an expression.

type system 14
The rules by which a language assigns and checks types.

U

UEFI / BIOS 6
The modern and older firmware interfaces that start the computer.

unauthorized access 1
Access to a system by someone who does not have permission.

Unicode 4
A standard that assigns a code to every character of the world's writing systems.

Unicode escape 14
A sequence such as A that stands for a Unicode character in source code.

universal Turing machine 5
A Turing machine that can simulate any other Turing machine.

unsigned 4
Describes a number type that has no negative values.

update 6
A change that installs a new version of software or fixes problems.

uppercase / lowercase 20
The two cases of letters, which can be changed with toUpperCase and toLowerCase.

URL 2
The address of a resource on the web.

usability 6
How easy and pleasant a system is to use.

USB 3
A standard connection for peripherals and for charging devices.

USB flash drive 3
A small portable storage device with flash memory.

user account 7
A set of information and rights for a person who uses the computer.

User Account Control (UAC) 7
A Windows feature that asks for permission before a program makes changes that need administrator rights.

user interface 6
The part of a system through which a user interacts with it.

user mode 6
The restricted mode in which applications run.

UTF-8 4
A variable-length Unicode encoding that uses one to four bytes per character.

utility 6
A small program that performs a maintenance task, such as disk cleanup.

V

varargs (variable-length arguments) 18
A method parameter that accepts any number of arguments, written with three dots.

variable 4
A named place in memory that holds a value that can change.

VBA 8
Visual Basic for Applications, the language for writing macros in Office.

verification / validation 12
Checking that a result is correct and that the right problem is solved.

version 14
A numbered release of a program or file.

version control 11
A system, such as Git, that records changes to files over time.

version history 8
A Word feature that lists the earlier saved versions of a document.

video memory (VRAM) 3
The memory on a graphics card used for images and textures.

view 8
A way of showing a document, such as Print Layout or Read Mode.

viewing angle 3
The angle from which a display can be seen without distortion.

virtual address 6
The address that a program uses before the memory system translates it.

virtual desktop 7
One of several separate desktops in Windows that organize open windows.

virtual machine 14
A program that runs like a computer, such as the JVM.

virtual machine (VM) 6
A software-based computer that runs as a program on a physical computer.

virtual memory 6
A technique that gives programs more memory than the physical RAM by using the disk.

virtualization 6
Creating virtual versions of hardware, such as several virtual machines on one computer.

virus 2
Malware that attaches itself to files and spreads when they are opened.

visual hierarchy 10
The use of size, color and position to show which content matters most on a slide.

volatile / temporary 3
Memory that loses its contents when power is turned off.

von Neumann architecture 4
A computer design in which instructions and data share one memory.

von Neumann bottleneck 4
The limit on speed caused by the single path between CPU and memory.

VPN 2
A virtual private network that encrypts traffic over a public network.

W

waiting time 6
The total time that a process waits in the ready queue.

waterfall model 11
A development model in which the stages follow one another in a fixed order.

watermark 8
Faint text or an image displayed behind the content of a page.

web browser 2
A program, such as Chrome, used to view web pages.

web crawler 2
A program that browses the web automatically to collect pages for a search index.

whitespace 20
Characters such as space, tab and newline that appear as blanks.

Wi-Fi 2
A wireless networking technology for connecting to a local network.

wide area network (WAN) 2
A network that covers a large area, such as a country, like the internet.

widening conversion 15
Automatic conversion of a smaller numeric type to a larger one, such as int to double.

widescreen 10
A display with a wide aspect ratio, such as 16:9.

wildcard 7
A character, such as * or ?, that stands for any characters in a search.

window 7
A rectangular area on the screen in which a program shows its content.

word 8
A unit of text, a group of letters between spaces, or the unit of data of a processor.

word count 8
The number of words in a document, shown in the status bar.

word processor 6
A program for writing and formatting documents, such as Microsoft Word.

workbook 9
An Excel file that contains one or more worksheets.

workload 3
The amount of work that is given to a system.

worksheet 9
A single sheet of cells in an Excel workbook.

World Wide Web 2
The system of linked pages and resources that is reached with browsers over the internet.

worm 2
Malware that copies itself over a network without needing a host file.

worst case 12
The input for which an algorithm takes the most steps.

wrap text 9
An Excel setting that shows long text on several lines within a cell.

write once, run anywhere 14
The Java principle that compiled programs run on any platform with a JVM.

WYSIWYG (what you see is what you get) 8
A kind of editing in which what is shown on screen matches the final printed result.

Z

zoom 8
To enlarge or reduce the display of a document or window.

APPENDIX B

Prerequisites for Computer Fundamentals and Introduction to Programming

Computer Fundamentals and Introduction to Programming · Prerequisites

The course does not expect any programming experience. This page collects the everyday computer skills and school mathematics on which the 21 presentations in the 12 chapters rest. Each topic card says what you should know, in which presentations it is used, and has two short self-check questions. The details open with the **Details and self-check** button, and clicking a question shows its answer.

The essential core. Typing accurately and being able to find your files, the four operations with integer quotient and remainder (`/` and `%`), percentages and averages, using letters in place of numbers, and/or/not, and following steps in order. Java itself is taught from scratch in Chapter 5. If something here is missing, a short review in the first weeks is enough.

Terms. The terms used in the slides are collected in the course glossary.

Computer use basics

Keyboard and mouse, careful typing, files and folders, the web and email, installing software. Chapters 1–3 build on these, and the Java labs take them for granted.

Keyboard, mouse and shortcuts

Clicking · selecting · Ctrl shortcuts

You should be able to select, copy, move and undo quickly with the keyboard. The Windows and Office presentations introduce many more shortcuts on top of these.

CHAPTERS Chapter 1.2 · Computer Hardware
Chapter 2.2 · The Windows Operating System
Chapter 3.1 · Microsoft Word
Chapter 3.2 · Microsoft Excel
Chapter 3.3 · Microsoft PowerPoint

- Mouse: a single click selects, a double click opens, and a right click opens the context menu. Dragging moves an item; holding Ctrl while dragging copies it.
- Selecting text: Shift with the arrow keys extends the selection. Ctrl + A selects everything, and a double click selects a single word.
- Basic shortcuts: Ctrl + C (copy), Ctrl + X (cut), Ctrl + V (paste), Ctrl + Z (undo), Ctrl + Y (redo), Ctrl + S (save), Ctrl + F (find).
- Window and system shortcuts used in Chapter 2: Win + E (File Explorer), Win + D (show the desktop), Alt + Tab (switch between windows), Win + Shift + S (capture a region of the screen).

TEST YOURSELF

▶ **With a paragraph selected, you press Ctrl + X and then Ctrl + V somewhere else. What happens to the paragraph?**

▶ **A sentence was deleted by accident. Which shortcut brings it back?**

Aa

Accurate typing and special characters

Upper and lower case · punctuation · brackets · keyboard layouts

Java is unforgiving: a single wrong capital letter or a missing semicolon stops compilation. Code must be typed letter for letter, including symbols that can be awkward to type on your keyboard layout.

CHAPTERS **Chapter 2.2 · The Windows Operating System** **Chapter 5 · Getting Started with Java** **Chapter 6 · Variables, Data Types and Operators** **Chapter 7 · Conditionals** **Chapter 11 · Strings**

- Java is case-sensitive: `System`, `system` and `SYSTEM` are three different names, and only the first one is defined.
- Symbols typed every day: `; { } ( ) [ ] " ' + - * / % = < > ! & |`.
- On a US keyboard the braces and brackets are on the keys next to P (`{ }` are Shift + `[ ]`). Many other layouts need AltGr for them, for example AltGr + 7 and AltGr + 0 for `{` and `}` on several European layouts. Switching between keyboard layouts (Win + Space) is explained in the Windows presentation.
- Look-alike characters must not be confused: the digit 0 and the letter O, the digit 1 and the letter l, the straight quote `"` and the curly quotes “ ” that Word inserts. Java does not accept curly quotes.

- Letter case depends on the locale: the Turkish dotted and dotless i can cause trouble in identifiers and string comparisons (Chapter 11, "Turkish Characters and Locale").

TEST YOURSELF

▶ **Does Java accept the line `system.out.println("Hi");` ?**

▶ **On a US keyboard, how are `{` and `}` typed?**



Files, folders, paths and extensions

Folder tree · absolute and relative paths · .java/.class

Programs are files that live in folders. You need to know where a file is saved, its full name including the extension, and how to read a path such as `C:\Course\Week1\Hello.java`.

CHAPTERS **Chapter 1.2 · Computer Hardware** **Chapter 2.1 · Software and Operating Systems** **Chapter 2.2 · The Windows Operating System** **Chapter 3.1 · Microsoft Word** **Chapter 5 · Getting Started with Java**

- A path lists the folders from the drive (the root) down to the file: drive `C:`, folders `Course` and `Week1`, file `Hello.java`.
- The extension is the part of the name after the last dot and shows the type of the file: `.docx`, `.xlsx`, `.pptx`, `.pdf`, `.txt`, `.java` (source code), `.class` (compiled bytecode). To make extensions visible, turn on the "File name extensions" option in File Explorer.
- A relative path starts from the current folder: `.` is the current folder and `..` is its parent folder.
- A Java rule (Chapter 5): the file that contains `public class Hello` must be named exactly `Hello.java`.
- Keeping the coursework in a separate folder for each week or chapter makes things easier. Saving everything to the Desktop or Downloads creates clutter.

TEST YOURSELF

▶ **What is the extension of the file at `C:\Users\Ann\Documents\homework\Hello.java`, and in which folder is it?**

▶ **If the current folder is `C:\Course\Week1`, where does the relative path `..\picture.png` point?**

@ Web browser, search and email

Web addresses · searching · attachments · safe habits

Throughout the course you download material, search for error messages, and submit assignments by email or through a learning platform. Chapter 1 also explains how web addresses (URLs), the Domain Name System (DNS) and email work behind the scenes.

CHAPTERS **Chapter 1.1 · Basic Concepts** **Chapter 1.4 · The Internet** **Chapter 1.5 · Computation and Computational Thinking** **Chapter 11 · Strings**

- A URL is read from left to right: first the protocol (`https`), then the domain name (`www.example.edu`), and last the path on the server (`/announcements`).
- For an effective search, type the error message in quotation marks exactly as shown, add the name of the language (“java”), and give priority to the official documentation.
- An email address consists of a user name, `@` and a domain name. A good email has a clear subject, the attachment is checked before sending, and the reply goes to the right person.
- Safe habits: a separate, strong password for each account, multi-factor authentication, and caution with unexpected links and attachments (phishing).
- Chapter 11 splits an email address into user name and domain name with `indexOf('@')` and `substring`. Knowing the structure of an address beforehand is therefore useful.

TEST YOURSELF

▶ **In the address `https://www.example.edu/en/announcements`, which part is the domain name?**

▶ **In the address `ann.lee@students.example.edu`, what are the parts before and after the `@` sign?**

⬇ Installing and updating software

Official sources · installers · versions

Before the programming chapters, a JDK (Java Development Kit) and an editor must be installed. To do this, the installer is downloaded from the official source, run, and the installed version is checked.

CHAPTERS **Chapter 2.1 · Software and Operating Systems** **Chapter 2.2 · The Windows Operating System** **Chapter 5 · Getting Started with Java**

- Programs should be downloaded only from the maker's own site or an official store. Stay away from “free download” sites and cracked software.
- You need to know your system: the Settings → System → About screen shows the Windows version and whether the system is 64-bit.
- After installation, verify it: open the program (or a terminal) and look at the version it reports.
- Keep the operating system and the browser up to date. Remove programs through Settings → Apps, not by deleting their folders.

TEST YOURSELF

▶ **In Windows 11, where can you see whether the system is 64-bit?**

▶ **Why is it wrong to uninstall a program by deleting its folder inside `C:\Program Files`?**

Arithmetic and numbers

Operator precedence, integers and decimal numbers, quotient and remainder, percentages and averages, powers and number bases, unit prefixes and logarithms.

Arithmetic and operator precedence

Parentheses · × ÷ first, then + – · left to right

Excel formulas and Java expressions follow the rules learned at school: parentheses first, then multiplication and division, and addition and subtraction last. Operations of the same precedence are performed from left to right.

CHAPTERS **Chapter 3.2 · Microsoft Excel** **Chapter 6 · Variables, Data Types and Operators** **Chapter 7 · Conditionals**

- Order of precedence: ( ) → exponentiation (`^` in Excel; Java has no power operator, so `Math.pow` is used) → `* / %` → `+ -`.

- Operations at the same level are done from left to right: `12 / 3 * 2` is (12 / 3) · 2 = 8, not 12 / 6.
- When in doubt, add parentheses. Parentheses cost nothing and show your intent clearly.
- Unary minus and comparisons also appear in the precedence table. Chapter 3 (Excel) and Chapter 6 (Java) give both tables.

Example (Chapter 6). In Java, the expression `"Result: " + 2 + 3` is evaluated from left to right and gives the text `Result: 23`. `"Result: " + (2 + 3)`, on the other hand, gives `Result: 5`.

TEST YOURSELF

▶ **What are `2 + 3 * 4` and `(2 + 3) * 4`?**

▶ **Calculated from left to right, what is `10 - 4 - 3`?**

2,5 Integers and decimal numbers

Integer · decimal number · rounding · decimal separator

A computer stores integers and decimal numbers differently. It matters which type of number a calculation is done with: the result of `5 / 2` is 2 in Java but 2.5 on a calculator.

CHAPTERS **Chapter 1.3 · Computers** **Chapter 3.2 · Microsoft Excel** **Chapter 6 · Variables, Data Types and Operators** **Chapter 7 · Conditionals**

- Integers (..., −2, −1, 0, 1, 2, ...) are stored exactly. Java's `int` type holds values up to 2,147,483,647.
- Decimal numbers are stored approximately (Chapter 1, “Why Isn't 0.1 Exact?”). In Java, `0.1 + 0.2` prints `0.30000000000000004`.
- Rounding and truncation are different: 3.99 rounded to the nearest integer is 4, but truncated (dropping the fractional part) it is 3. In Java, `(int) 3.99` truncates and gives 3.
- Notation: many countries write 3,14 with a comma, and spreadsheets set to those regional settings do too. In Java source code, a period is always used: `3.14`.

TEST YOURSELF

▶ **What are `5 / 2` and `5.0 / 2` in Java?**

▶ **What do 7.5 and 7.49 become when rounded to the nearest integer and when truncated?**

%

Division with quotient and remainder

/ and % on integers · odd/even · digits

Division with a remainder, learned in primary school (17 ÷ 5 = 3, remainder 2), is the most-used mathematics in the course: `/` gives the quotient and `%` gives the remainder.

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking** **Chapter 4.1 · Algorithms** **Chapter 6 · Variables, Data Types and Operators** **Chapter 7 · Conditionals** **Chapter 8 · Loops** **Chapter 9 · Methods** **Chapter 12 · Recursion**

- For positive integers, a = b · q + r and 0 ≤ r < b. In Java you write `q = a / b` and `r = a % b`.
- Odd or even: if `n % 2 == 0`, n is even (Chapter 7 “Odd or Even?”, Chapter 9 `isEven()`).
- Digits: `n % 10` gives the last digit, and `n / 10` drops it. Repeating these two steps counts or adds up the digits (Chapter 8, Chapter 9, Chapter 12).
- Unit conversion: from seconds to hours, minutes and seconds (Chapter 6 “Seconds to Hours”).
- Divisibility: if `a % b == 0`, b divides a exactly (prime numbers, greatest common divisor, leap years).
- In Java the sign of the remainder is the same as the sign of the dividend: `-7 % 3` is −1 (Chapter 6).

TEST YOURSELF

▶ **With integers, what are `17 / 5` and `17 % 5`?**

▶ **Using / and %, how many hours, minutes and seconds is 3725 seconds?**

x̄

Percentages and averages

Percentage of a value · weighted grade · average

Grade sheets in Excel, methods that compute an average temperature, and array statistics rest on percentages, plain averages and weighted averages.

CHAPTERS **Chapter 3.2 · Microsoft Excel** **Chapter 6 · Variables, Data Types and Operators** **Chapter 9 · Methods** **Chapter 10 · Arrays**

- p% of x is x · p / 100. A p% increase multiplies the value by (1 + p/100), and a p% decrease by (1 − p/100).
- Average (arithmetic mean) = total ÷ count. In Excel you write `=AVERAGE(B2:B10)`. In Java a loop first adds up, then divides. Use `double` so that the fractional part is not lost.
- Weighted average: for a course grade where the midterm counts 40% and the final 60%, the grade is 0.4 · midterm + 0.6 · final.
- Chapter 10 finds the elements above the average, and the smallest and largest values, in an array.

TEST YOURSELF

▶ **Midterm 70 (weight 40%), final 85 (weight 60%). What is the course grade?**

▶ **If a $250 price gets a 20% discount, what is the new price? What is the average of 4, 8 and 9?**

10^n Powers and place value

Powers · position of digits · powers of 2

The value of a digit depends on the place it occupies (ones, tens, hundreds = 10^0, 10^1, 10^2). The same idea, with powers of 2, explains binary numbers. Computing powers is also a classic example for loops and recursion.

CHAPTERS **Chapter 1.1 · Basic Concepts** **Chapter 1.3 · Computers** **Chapter 3.2 · Microsoft Excel** **Chapter 8 · Loops** **Chapter 12 · Recursion**

- a^n is a multiplied by itself n times. $a^0 = 1$, $a^1 = a$ and $a^m \cdot a^n = a^{m+n}$.
- Place value: $307 = 3 \cdot 10^2 + 0 \cdot 10^1 + 7 \cdot 10^0$.
- Powers of 2 worth knowing: $2^0 = 1$, $2^1 = 2$, $2^4 = 16$, $2^8 = 256$, $2^{10} = 1024$.
- In Excel you write `=2^10`. In Java, `^` is not exponentiation but the XOR operation. For powers use `Math.pow(2, 10)` or a loop (Chapter 8 “Exponentiation”, Chapter 12 “Fast Power”).

TEST YOURSELF

▶ **What are 2^{10} and 5^0?**

▶ **Starting with x = 1, if x is doubled 4 times, what is x?**

01 Number bases, bits and bytes

Binary numbers · KB and KiB · limited ranges

All data is stored as bits. You need to convert small numbers between decimal and binary, work out storage sizes in bytes, and see why a fixed number of bits limits the range of values.

CHAPTERS **Chapter 1.1 · Basic Concepts** **Chapter 1.2 · Computer Hardware** **Chapter 1.3 · Computers** **Chapter 6 · Variables, Data Types and Operators** **Chapter 8 · Loops**

- In binary the place values go 1, 2, 4, 8, 16, …: $1011_2 = 8 + 0 + 2 + 1 = 11$.
- To convert from decimal to binary, divide the number by 2 repeatedly and read the remainders from bottom to top (Chapter 1, Chapter 8 “Decimal to Binary”).
- n bits give 2^n different values: 8 bits = 1 byte = 256 values (0–255 unsigned).
- Units: in the decimal (SI) notation that disk makers use, 1 KB = 1000 B, while in binary (IEC) notation 1 KiB = 1024 B. The table in Chapter 1 writes 1 KB = 1,024 B, and the units based on 1024 are called KiB, MiB and GiB.
- Network speeds are given in bits per second: 100 Mbit/s is 12.5 MB/s.
- A fixed-width value overflows: adding 1 to the 8-bit value 255 needs a ninth bit. Java's `int` wraps around after 2,147,483,647 (Chapter 6 “Overflow”).

TEST YOURSELF

▶ **What is 1011_2 in decimal, and how is 13 written in binary?**

▶ **About how many GiB does Windows show for a disk sold as 500 GB ($500 \cdot 10^9$ bytes)?**

GHz Units of measurement and prefixes

kilo, mega, giga · milli, micro, nano · Hz and seconds

In hardware, networking and operating systems, very large and very small numbers are written with prefixed units: 3 GHz is 3 billion cycles per second, and 5 ms is five thousandths of a second. You need to know which power of 10 each

prefix stands for, and to be able to go from a frequency to a duration.

CHAPTERS Chapter 1.2 · Computer Hardware | Chapter 1.3 · Computers | Chapter 1.4 · The Internet | Chapter 2.1 · Software and Operating Systems

- Prefixes that enlarge: kilo (k) = 10^3, mega (M) = 10^6, giga (G) = 10^9, tera (T) = 10^{12}. Prefixes that shrink: milli (m) = 10^{-3}, micro (µ) = 10^{-6}, nano (n) = 10^{-9}.
- A negative exponent means division: 10^{-3} = 1/1000. 1 s = 1000 ms = 1,000,000 µs = 10^9 ns.
- Hertz (Hz) is the number of repetitions per second. The duration of one cycle is the inverse of the frequency: on a 3 GHz processor, $1 \div (3 \times 10^9)$ s ≈ 0.33 ns.
- Keep the units the same in calculations: a 100 Mbit/s line carries 12.5 MB per second, so a 500 MB file downloads in 500 ÷ 12.5 = 40 s at best. If bits and bytes are confused, the result is off by a factor of 8.
- Bandwidth is the amount of bits carried per second; latency (ping) is the time for the first bit to reach the other end, measured in ms. One can be high without the other being good.

TEST YOURSELF

▶ **How long does one clock cycle take on a 2 GHz processor, in ns?**

▶ **How many milliseconds is 0.25 s? How many seconds is 20 ms?**

Logarithm ($\log_2$)

Inverse of powers of 2 · number of halvings · number of bits

$\log_2 n$ tells you which power of 2 equals n: since 2^{10} = 1024, $\log_2 1024 = 10$. The same number also gives how many times n can be halved until it reaches 1. The number of bits needed to encode a set of values, and the number of rounds of a binary search, are found this way.

CHAPTERS Chapter 1.1 · Basic Concepts | Chapter 4.1 · Algorithms | Chapter 4.1 · Algorithms | Chapter 10 · Arrays | Chapter 12 · Recursion

- It is the inverse of the exponent: if $2^k = n$ then $\log_2 n = k$. $\log_2 1 = 0$, $\log_2 8 = 3$, $\log_2 1024 = 10$.
- For numbers that are not exact powers, the result falls between two integers: since $2^5 = 32 < 58 < 64 = 2^6$, $\log_2 58 \approx 5.86$. The number of bits needed to encode N different values is this value rounded up: 6 bits for 58 values.

- In a sorted array of n elements, binary search halves the range in every round and finishes in at most $\lfloor \log_2 n \rfloor + 1$ rounds. For a million elements, that bound is 20 rounds.
- O(log n) and O(n log n) in big-O notation describe this slow growth: when n grows a thousandfold, $\log_2 n$ grows by only about 10, because $2^{10} \approx 1000$.

TEST YOURSELF

▶ **What is $\log_2 64$? In a sorted array of 64 elements, in at most how many rounds does binary search finish?**

▶ **At least how many bits are needed to give each of 40 students a distinct binary number?**

Algebra, logic and functions

Letters standing in for numbers, simple formulas, inequalities, and/or/not, and the function as a rule that turns input into output.

x

Variables and basic algebra

Letters for numbers · substitution · assignment

In algebra, a letter holds the place of a number. In programming, a variable is a named box whose contents can change. Substituting values for the letters in an expression is exactly what the computer does.

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking** **Chapter 4.3 · Introduction to Programming** **Chapter 4.2 · Flowcharts** **Chapter 6 · Variables, Data Types and Operators** **Chapter 8 · Loops**

- Substitution: for a = 3 and b = 4, $a^2 + b^2 = 9 + 16 = 25$.
- In Java, `=` is not equality but assignment: `x = x + 1` means “compute x + 1 and write the result back into x”. It is not a false equation. Equality is tested with `==`.
- Swapping two values needs a temporary variable: `t = a; a = b; b = t;` (Chapter 6 “Swapping”, Chapter 9 “swap Pitfall”).
- Counters and accumulators (Chapter 4 flowcharts, Chapter 8): `count = count + 1`, `total = total + x`.

TEST YOURSELF

▶ **With x = 5, if `x = x + 1;` runs first and then `x = x * 2;`, what is x?**

▶ **With a = 3 and b = 5, after `t = a; a = b; b = t;` runs, what are a and b?**

πr²

Everyday formulas

Area · temperature · quadratic equation

Many first programs turn a school formula into code: the area and circumference of a circle, converting Celsius to Fahrenheit, the roots of a quadratic equation, absolute value.

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking** **Chapter 4.3 · Introduction to Programming** **Chapter 6 · Variables, Data Types and Operators** **Chapter 7 · Conditionals** **Chapter 9 · Methods**

- Circle: area πr^2, circumference $2\pi r$ (Chapter 6 “Area and Circumference of a Circle”, `Math.PI`).
- Temperature: $F = C \cdot 9/5 + 32$ and $C = (F - 32) \cdot 5/9$ (Chapter 6, Chapter 9 “Temperature Methods”).
- The quadratic equation $ax^2 + bx + c = 0$: the discriminant is $\Delta = b^2 - 4ac$. If $\Delta > 0$ there are two real roots, if $\Delta = 0$ there is one, and if $\Delta < 0$ there are no real roots. The roots are found with $(-b \pm \sqrt{\Delta}) / 2a$ (Chapter 7).
- The absolute value $|x|$ equals x if $x \geq 0$, otherwise $-x$ (Chapter 7 “Absolute Value”). The quantity under a square root cannot be negative (Chapter 1 “Square Root, Step by Step”).

TEST YOURSELF

▶ **How many degrees Fahrenheit is 25 °C?**

▶ **For $x^2 - 5x + 6 = 0$, what are Δ and the roots?**

<

Inequalities and ranges

< ≤ > ≥ · ranges · bounds

Conditions such as “10 < x < 20”, “grade ≥ 50” or “index from 0 to n − 1” are inequalities. Whether an endpoint is included is the source of many off-by-one errors.

CHAPTERS **Chapter 3.2 · Microsoft Excel** **Chapter 4.1 · Algorithms** **Chapter 4.2 · Flowcharts** **Chapter 7 · Conditionals** **Chapter 8 · Loops** **Chapter 10 · Arrays**

- A strict inequality (<, >) leaves the endpoint out, while a non-strict one (≤, ≥) includes it. In Java, ≤ is written `<=` and ≠ is written `!=`.
- A two-sided inequality is two conditions joined by "and": 10 < x < 20 becomes `x > 10 && x < 20`. In Java you cannot write `10 < x < 20`.
- Counting the integers in a range: from a to b (both included) there are b − a + 1 numbers.
- Loops: `for (int i = 0; i < 5; i++)` runs 5 times (i = 0..4), while with `i <= 5` it runs 6 times (Chapter 8 "Off-by-One Errors"). An array of length n has indices 0 … n − 1 (Chapter 10).

TEST YOURSELF

▶ **How many integers satisfy 10 < x < 20? How many satisfy 10 ≤ x ≤ 20?**

▶ **How many times does the loop `for (int i = 0; i < 5; i++)` run? How many times if it is `i <= 5`?**

Logic: and, or, not

True/false · truth tables · De Morgan

Every decision in a program is a statement that is true or false, and it is often combined with and, or, not. This logic appears in logic gates (Chapter 1), in Excel's AND/OR/NOT functions (Chapter 3) and in Java's `&&`, `||` and `!` operators.

CHAPTERS **Chapter 1.1 · Basic Concepts** **Chapter 1.3 · Computers** **Chapter 3.2 · Microsoft Excel** **Chapter 6 · Variables, Data Types and Operators** **Chapter 7 · Conditionals** **Chapter 8 · Loops**

- A and B is true only if both are true. A or B is true if at least one is true. Not A reverses the value of A.
- A truth table lists all the cases: with two inputs there are 4 rows.
- De Morgan: not (A and B) = (not A) or (not B); not (A or B) = (not A) and (not B) (Chapter 7).
- The "or" of logic is inclusive: true or true is true. The exclusive or (XOR), meaning "one or the other but not both", is a separate operation.

- The leap-year rule (Chapter 7): divisible by 4 and (not divisible by 100 or divisible by 400).

TEST YOURSELF

▶ **With x = 15, what are the values of `x > 10 && x < 20` and `!(x > 10)`?**

▶ **According to the leap-year rule, are 1900 and 2000 leap years?**

f(x)

Functions as input → output

f(x) · arguments · return value · composition

A function in mathematics takes inputs and gives a single output: f(x) = 2x + 3. Excel functions (`=SUM(…)`) and Java methods work the same way: arguments go in, a value comes back.

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking** **Chapter 3.2 · Microsoft Excel** **Chapter 4.3 · Introduction to Programming** **Chapter 9 · Methods** **Chapter 12 · Recursion**

- f(x) = 2x + 3 means “given x, return 2x + 3”, and f(4) = 11. The name of the input does not matter: f(t) = 2t + 3 is the same function.
- Functions with several inputs: max(a, b), gcd(a, b), and in Excel `=MAX(B2:B10)`.
- Composition: when computing f(f(1)), the inner call is evaluated first. Chapter 9 follows nested calls step by step on the call stack.
- A Java method has a parameter list (the inputs), a return type (the type of the output) and a body. `void` methods do not return a value.

TEST YOURSELF

▶ **If f(x) = 2x + 3, what are f(4) and f(f(1))?**

▶ **In Excel, cells B2:B4 contain 3, 9 and 4. What do `=MAX(B2:B4)` and `=SUM(B2:B4)` return?**

Sequences, divisibility and recursion

Sums such as 1 + 2 + … + n, prime numbers and the greatest common divisor, factorial and Fibonacci, counting layouts of rows and columns and positions. These are the standard examples of the presentations on loops, methods, arrays and recursion.

Σ Number sequences and sums

1 + 2 + ... + n · digit sum · running total

Adding up a sequence of numbers term by term is the first job of every loop (Chapter 4 flowchart "Example: Sum from 1 to n", Chapter 8, Chapter 10, Chapter 12). Knowing the closed-form formulas lets you check the answer a program gives.

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking** **Chapter 4.2 · Flowcharts** **Chapter 8 · Loops** **Chapter 10 · Arrays** **Chapter 12 · Recursion**

- 1 + 2 + ... + n = n(n + 1)/2 (Gauss). For example, 1 + 2 + ... + 100 = 5050.
- Arithmetic sequence: starts at a and increases by d (2, 4, 6, ...). Geometric sequence: starts at a and is multiplied by r at each step (1, 2, 4, 8, ...).
- A running sum starts at 0 (the identity element of addition), and a running product starts at 1.
- The digit sum of 4827 is 4 + 8 + 2 + 7 = 21 (Chapter 8 "Sum of Digits", with the recursive version in Chapter 12).

TEST YOURSELF

▶ **What are the sums 1 + 2 + ... + 100 and 2 + 4 + ... + 20?**

▶ **With s = 0, if s = s + i · i is done for i = 1, 2, 3, 4, what is s at the end?**

÷ Divisibility, primes and GCD

Divisors · primality test · Euclid · LCM

Primality testing, perfect numbers, GCD/LCM and the sieve of Eratosthenes are standard examples in the presentations on algorithms, loops, methods, arrays and recursion.

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking** **Chapter 4.1 · Algorithms** **Chapter 8 · Loops** **Chapter 9 · Methods** **Chapter 10 · Arrays** **Chapter 12 · Recursion**

- If n % d == 0, d divides n. A prime number has exactly two divisors: 1 and itself. 1 is not prime, and 2 is the only even prime.
- To find out whether n is prime, it is enough to try the divisors up to √n (Chapter 4, Chapter 8, Chapter 9 `isPrime()`).

- Euclid's algorithm: apply gcd(a, b) = gcd(b, a % b) until the remainder is 0 (Chapter 1, Chapter 9 `gcd()`, Chapter 12). Then lcm(a, b) = a · b / gcd(a, b).
- A perfect number equals the sum of its divisors other than itself: 6 = 1 + 2 + 3, 28 = 1 + 2 + 4 + 7 + 14 (Chapter 8).

TEST YOURSELF

▶ **Is 91 prime?**

▶ **Using Euclid's algorithm, what are gcd(48, 18) and then lcm(48, 18)?**

n!

Factorial, Fibonacci and recursive definitions

n! = n · (n − 1)! · base case · rapid growth

Some sequences are defined by referring to earlier terms: n! = n · (n − 1)! with 0! = 1, and each Fibonacci number is the sum of the two before it. The recursive methods in Chapter 12 are written in exactly this way.

CHAPTERS **Chapter 4.1 · Algorithms** **Chapter 8 · Loops** **Chapter 12 · Recursion**

- Since n! = 1 · 2 · … · n, 5! = 120. The recursive definition is n! = n · (n − 1)!, and the base case is 0! = 1 (1! = 1 also holds).
- Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, … Each term is the sum of the two terms before it.
- Every recursive definition must have a base case that stops it. Without a base case the computation never ends (Chapter 12 `StackOverflowError`).
- These numbers grow quickly: 13! = 6,227,020,800 does not fit in an `int` (the largest value is 2,147,483,647). Chapter 12 therefore uses `long` and discusses overflow.

TEST YOURSELF

▶ **Given that 5! = 120, what is 6! by the rule n! = n · (n − 1)!?**

▶ **What Fibonacci number comes after 5, 8, 13?**

▦

Counting, row-column layouts and positions

Rows × columns · patterns · counting from 0

Nested loops, star patterns, multiplication tables, two-dimensional arrays and string indices all rest on the same two skills: counting the cells in rows and columns, and numbering positions from 0.

CHAPTERS **Chapter 3.2 · Microsoft Excel** **Chapter 4.2 · Flowcharts** **Chapter 8 · Loops** **Chapter 10 · Arrays** **Chapter 11 · Strings**

- A table with r rows and c columns has r · c cells. A nested loop whose outer loop takes r steps and inner loop takes c steps runs r · c times.
- Triangle patterns: a triangle whose rows have 1, 2, …, n stars has 1 + 2 + … + n stars in total (Chapter 8 “Growing Triangle”).
- Java counts positions from 0: an array or string of length n has indices 0 … n − 1, and the last one is n − 1 (Chapter 10, Chapter 11).
- Excel cell references (B3 = column B, row 3) and a multiplication table built with mixed references (Chapter 3) rest on the same row-column thinking.

TEST YOURSELF

▶ **The rows of a star triangle have 1, 2, 3, 4, 5 stars. How many stars are there in total?**

▶ **The word “PROGRAM” has 7 letters. If positions are numbered from 0, what is the last position, and which letter is at position 2?**

Reading and problem-solving skills

Following instructions literally, tracking values step by step, testing with carefully chosen inputs, and reading technical English.

123

Following step-by-step instructions

Recipes · algorithms · flowchart symbols

An algorithm is a recipe that must be carried out literally and in order, “without guessing what was meant”. Reading flowcharts and pseudocode is the bridge from everyday instructions to Java.

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking** **Chapter 4.3 · Introduction to Programming** **Chapter 4.1 · Algorithms** **Chapter 4.2 · Flowcharts**

- Each step is carried out in order and exactly as stated. A loop says “go back and repeat”, and a decision says “take only one of the branches”.
- Flowchart symbols (Chapter 4): oval = start/end, parallelogram = input/output, rectangle = process, diamond = decision, arrows = order.
- There are three building blocks: sequence, selection (if) and repetition (loop) (Chapter 1, Chapter 7).
- A good instruction is clear and ends in a finite number of steps. Chapter 4 lists the properties of a good algorithm.

TEST YOURSELF

▶ **Steps: (1) x ← 1. (2) Repeat 4 times: x ← x · 2. (3) Print x. What is printed on the screen?**

▶ **Which shape represents a decision in a flowchart, and how many arrows leave it?**

Tracing with a table of values

Trace tables · the state after each step

Every presentation has “Step by Step” slides that show the values of the variables after each step. You should be able to keep such a table on paper too: a column for each variable and a row for each step.

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking** **Chapter 4.1 · Algorithms** **Chapter 4.2 · Flowcharts** **Chapter 8 · Loops** **Chapter 9 · Methods** **Chapter 12 · Recursion**

- The variable names go in the column headings. Each time a value changes, a new row is added, and no step is skipped.
- The loop condition is checked before every round. The final check that ends the loop counts too.
- The table is compared with the program's output. The first row where the two differ shows where the bug is.
- Chapter 9 and Chapter 12 carry tracing over to the call stack: each method call gets its own row (frame).

TEST YOURSELF

▶ **a = 10, b = 3, k = 0. As long as a ≥ b, do a ← a − b and k ← k + 1. What are a and k at the end?**

▶ **In the previous question, how many times is the condition a ≥ b checked?**

✓

Breaking a problem into parts and testing

Decomposition · test cases · edge cases

Before writing code, the problem is split into inputs, steps and outputs. After the code is written, it is tested both with ordinary inputs and with the boundary values at which programs most often go wrong.

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking** **Chapter 4.3 · Introduction to Programming** **Chapter 4.1 · Algorithms** **Chapter 4.2 · Flowcharts** **Chapter 8 · Loops** **Chapter 9 · Methods**

- The four elements of computational thinking (Chapter 1): decomposition, pattern recognition, abstraction and algorithm design.
- Before code is written, the input and output are fixed with an example (Chapter 1 “Algorithm Design and Testing”, Chapter 9 “Contract First”).
- Test cases: an ordinary value, the boundary values on either side of every condition, and special values such as 0, 1, negative numbers and empty input.
- Change only one thing at a time and run the program again. Inputs that exposed a bug are kept as permanent tests.

TEST YOURSELF

▶ **The rule is “passes if the grade is ≥ 50”. Which two inputs test the boundary?**

▶ **Which four years could test all branches of the leap-year rule together?**

EN

Technical English and error messages

Java keywords · javac and runtime messages

Java's keywords, method names and all the messages from the compiler and the runtime environment are in English. Reading these messages carefully (what, where, which line) is the fastest way to fix a program.

CHAPTERS **Chapter 5 · Getting Started with Java** **Chapter 6 · Variables, Data Types and Operators** **Chapter 7 · Conditionals** **Chapter 8 · Loops** **Chapter 9 · Methods** **Chapter 10 · Arrays** **Chapter 11 · Strings** **Chapter 12 · Recursion**

- Keywords are ordinary English words with a precise meaning: `if`, `else`, `while` ("as long as"), `for`, `return` ("give back"), `break`, `new`, `void`, `static`.
- javac messages give the file, the line, a short explanation and a caret (^) marking the place of the error: `Hello.java:5: error: ';' expected`.
- Common messages: `cannot find symbol` (a misspelled or undeclared name), `incompatible types`, `missing return statement`.
- Runtime messages are exceptions: `ArithmeticException: / by zero`, `ArrayIndexOutOfBoundsException`, `StackOverflowError`.
- Fix the first error first, because later messages are often a consequence of it (Chapter 5 "Common Compile Errors").

TEST YOURSELF

▶ **javac gives the message `error: ';' expected`. What should you look at?**

▶ **What does the message `error: cannot find symbol … symbol: variable totl` usually mean?**

Tools and setup

A JDK and an editor or IDE, a few terminal commands, and an office suite.



A JDK and an editor or IDE

javac · java · .java → .class

From Chapter 5 on, Java programs are written, compiled and run on your own computer. For this, install a current JDK and either a plain editor or an IDE (for example Eclipse, IntelliJ IDEA or VS Code with a Java extension).

CHAPTERS **Chapter 4.3 · Introduction to Programming** **Chapter 5 · Getting Started with Java**

- Install the JDK, not just the JRE (Java Runtime Environment). The messages in Chapter 5 were produced with JDK 21. Check the installation with `javac -version` and `java -version`.

- Source code is a plain-text file with the extension `.java`. Word documents cannot be compiled.
- The workflow is as follows: write `Hello.java` → the command `javac Hello.java` produces the file `Hello.class` (bytecode) → the command `java Hello` runs it on the Java Virtual Machine (JVM).
- An IDE does these steps with one button and underlines errors as you type. You should still do these steps by hand at least once.
- Save files as UTF-8 so that non-ASCII characters in strings are not garbled (`javac -encoding UTF-8`, Chapter 5).

TEST YOURSELF

▶ **With which command is `Hello.java` compiled, with which is the program run, and which file does compilation create?**

▶ **The file is named `Homework1.java` but contains `public class Hello`. What happens?**

>_

Using the terminal (at the most basic level)

cd · dir/ls · running commands

A few commands are enough: seeing where you are, listing files, changing folders, and running `javac` / `java`. Chapter 2 introduces these commands for Windows (cmd) and for Linux/macOS.

CHAPTERS **Chapter 2.1 · Software and Operating Systems** **Chapter 2.2 · The Windows Operating System** **Chapter 5 · Getting Started with Java**

- Opening a terminal: Start → “cmd” (Command Prompt) or Windows Terminal. If you type `cmd` in File Explorer's address bar, the terminal opens in that folder.
- Windows cmd: `cd` (show or change the folder), `dir` (list), `mkdir` (new folder). Linux/macOS: `pwd`, `ls`, `cd`, `mkdir`.
- `cd ..` goes up to the parent folder. Paths that contain spaces are written in quotation marks: `cd "C:\Course Notes"`.
- Commands run in the current folder: if you are in the wrong folder, `javac Hello.java` gives a “file not found” error.
- `del` and `rm` delete files permanently (they do not send them to the Recycle Bin). Check the command before pressing Enter.

TEST YOURSELF

▶ **If you are in the folder `C:\Course\Week1` and type `cd ..`, which folder do you move to?**

▶ **What is the Windows command that lists the files in the current folder, and what is the Linux/macOS equivalent?**

Office software

Word · Excel · PowerPoint or a free alternative

Chapter 3 is taught with the English-language interface of Microsoft Word, Excel and PowerPoint. The university's Microsoft 365 account or a free alternative such as LibreOffice is enough to follow the course.

CHAPTERS **Chapter 1.1 · Basic Concepts** **Chapter 3.1 · Microsoft Word** **Chapter 3.2 · Microsoft Excel** **Chapter 3.3 · Microsoft PowerPoint**

- You should know how to open and save files (choosing the folder and format with “Save As”), print, and export as PDF.
- Formats: `.docx` (Word), `.xlsx` (Excel), `.pptx` (PowerPoint). Free suites can also open and save these files.
- Excel function names and argument separators depend on the language and regional settings of the installation. English Excel writes `=IF(B2>=50,"Pass","Fail")` with commas; some regional settings use semicolons instead.
- Basic Excel vocabulary: cell (B3), range (B2:B10), formula (starts with =), relative and absolute reference (`$A$1`).

TEST YOURSELF

▶ **What is the Excel formula that adds the cells from A1 to A10?**

▶ **Which extensions do Word, Excel and PowerPoint use by default?**

No prerequisite matches the search or filters.

▶ Presentation map

© Sercan Külcü · Computer Fundamentals and Introduction to Programming

APPENDIX C

Important People in Computing and Programming

Computer Fundamentals and Introduction to Programming · People

Short profiles of the 74 mathematicians, engineers and programmers behind the ideas used in the course, collected on 58 cards. The list runs from Euclid's algorithm and Boolean logic to Turing machines, the transistor, the Internet, Unix, spreadsheets and Java. Each card links to the chapters of the course in which the person's work appears; the details open with the button at the bottom of the card.

EU

Euclid of Alexandria

c. 300 BCE · Greek · Alexandria

Euclid's algorithm (GCD) · *Elements*, Book VII

In his *Elements* he described how to find the greatest common divisor (GCD) of two numbers. The method is one of the oldest known algorithms and is still taught today.

greatest common divisor | remainder | termination

CHAPTERS **Chapter 9 · Methods** **Chapter 12 · Recursion**

Why it matters: The rule gcd(a, b) = gcd(b, a mod b) stops when b = 0. In the course it is the model example for both the loop version and the recursive version of a method.

Euclid's original method uses repeated subtraction; the remainder form is its modern equivalent. The numbers shrink at every step, which is the standard argument that the algorithm always stops.

ER

Eratosthenes of Cyrene

c. 276 – c. 194 BCE · Greek · Library of Alexandria

Sieve of Eratosthenes · prime numbers

He was the chief librarian of the Library of Alexandria. The method of finding primes by repeatedly marking off the multiples of each prime is named after him.

prime numbers | sieve of Eratosthenes | boolean array

CHAPTERS **Chapter 8 · Loops** **Chapter 10 · Arrays**

Why it matters: The sieve keeps track of whether each number has been crossed out in a `boolean` array. Once a prime is found, all of its multiples are marked in a single pass. The outer loop can stop when $i \cdot i > n$, because every composite number up to n has a prime divisor no larger than $\sqrt{n}$. This is much faster than testing each number by trial division.

He calculated the circumference of the Earth by comparing the noon shadows at Alexandria and Syene (modern Aswan) at the summer solstice. Some historians think Eratosthenes did not invent the method and only gave the name "sieve".

AK

Muhammad ibn Musa al-Khwarizmi

c. 780 – c. 850 · Persian · House of Wisdom, Baghdad

Origin of the words "algorithm" and "algebra"

He wrote influential books on arithmetic with Hindu-Arabic numerals and on solving equations. The Latin form of his name, Algoritmi, became the word "algorithm".

step-by-step procedures | decimal numerals | algebra

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking** **Chapter 4.1 · Algorithms**

Why it matters: The central idea of the course carries his name: a finite, precise sequence of steps that solves a problem.

The word *algebra* comes from *al-jabr* in the title of his book on equations (c. 820). Latin translations of his arithmetic book spread the positional decimal notation to medieval Europe.

LF

Leonardo of Pisa (Fibonacci)

c. 1170 – c. 1250 · Italian · Pisa

***Liber Abaci* · 1202**

His *Liber Abaci* (1202) popularised Hindu-Arabic numerals in Europe. The rabbit problem in the book produces the sequence 0, 1, 1, 2, 3, 5, 8, ...

Fibonacci sequence | recurrence relation | iteration and recursion

CHAPTERS **Chapter 4.1 · Algorithms** **Chapter 8 · Loops** **Chapter 12 · Recursion**

Why it matters: $F(n) = F(n-1) + F(n-2)$ is the classic loop exercise. The sequence also shows why naive recursion repeats the same work again and again.

The sequence was already known in Indian mathematics. The name "Fibonacci sequence" was popularised by Édouard Lucas in the 19th century. The terms grow so fast that even Java's `long` type overflows after the 92nd term.

GL

Gottfried Wilhelm Leibniz

1646–1716 · German · polymath, Hanover

Binary arithmetic · 1703 · stepped reckoner

He published binary arithmetic, which uses only 0 and 1, in 1703. He also built a mechanical calculator called the stepped reckoner.

binary numbers | mechanical calculation | symbolic reasoning

CHAPTERS **Chapter 1.1 · Basic Concepts** **Chapter 1.3 · Computers**

Why it matters: Every number, character and instruction in a computer is stored in the binary system Leibniz described.

His essay "Explication de l'arithmétique binaire" appeared in the yearbook of the Paris Academy of Sciences. Leibniz also dreamed of a formal language in which reasoning could be carried out by calculation.

JJ

Joseph Marie Jacquard

1752–1834 · French · weaver and inventor, Lyon

Punched-card loom · 1804

He built a loom mechanism that wove patterns automatically (first demonstrated around 1804). A chain of punched cards controlled the mechanism.

punched cards | stored instructions | automation

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking**

Why it matters: When the cards change, the machine stays the same and the pattern changes. This is an early form of keeping the program separate from the hardware.

Babbage planned to feed the Analytical Engine with Jacquard-style cards. Punched cards went on to carry data and programs until the 1970s.

CG

Carl Friedrich Gauss

1777–1855 · German · University of Göttingen

The sum 1 + 2 + … + n = n(n + 1)/2

According to a well-known (perhaps embellished) story, as a schoolboy he instantly found the sum of the numbers from 1 to 100. He did it by pairing terms from the beginning and the end.

closed-form formula | efficiency | loop vs. formula

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking**

Why it matters: In the course, a summing loop (n additions) is compared with Gauss's formula (one multiplication and one division). This shows that algorithms can have very different costs.

Gauss is one of the most important mathematicians in history. He worked on number theory, astronomy, geodesy and magnetism. Gaussian elimination and the Gaussian (normal) distribution carry his name.

CB

Charles Babbage

1791–1871 · English · University of Cambridge (Lucasian Professor)

Difference Engine · Analytical Engine (1837)

He designed the Difference Engine for computing tables. In 1837 he then designed the Analytical Engine, a general-purpose mechanical computer that was never completed.

store and mill | punched-card input | general-purpose machine

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking**

Why it matters: The Analytical Engine separated memory (the "store") from processing (the "mill"). Today's computers use the same arrangement.

The design included conditional branching and loops controlled by punched cards. In 1991 the Science Museum in London completed a working Difference Engine No. 2 from his drawings.

AM

Augustus De Morgan

1806–1871 · English · University College London

De Morgan's laws · *Formal Logic*, 1847

He stated as rules that the negation of an AND expression equals the OR of the negations of its terms, and the reverse. The laws appeared in a paper of 1846 and were published again in his book *Formal Logic* in 1847.

De Morgan's laws | logical operators | negating a condition

CHAPTERS **Chapter 7 · Conditionals**

Why it matters: In Java, `!(a && b)` always has the same value as `!a || !b`, and `!(a || b)` the same as `!a && !b`. When you negate a condition, the `!` is distributed into the parentheses and `&&` and `||` swap places: the condition `!(s >= 15 && s <= 30)` becomes `s < 15 || s > 30`.

He was born in Madurai, India, and was a professor of mathematics at University College London for many years. In 1840–1842 he taught Ada Lovelace mathematics by correspondence. He became the first president of the London Mathematical Society, founded in 1865.

GB

George Boole

1815–1864 · English · Queen's College Cork, Ireland

Boolean algebra · 1847, 1854

He treated logic as an algebra of true/false values. He developed the idea in his books *The Mathematical Analysis of Logic* (1847) and *The Laws of Thought* (1854).

AND / OR / NOT | truth tables | boolean type

CHAPTERS **Chapter 1.3 · Computers** **Chapter 6 · Variables, Data Types and Operators** **Chapter 7 · Conditionals**

Why it matters: The logic gates in a processor and the conditions in `if` statements and loops are all Boolean expressions. Java's `boolean` type carries his name.

Boole was largely self-taught and became the first professor of mathematics at Queen's College Cork. Claude Shannon later showed that Boolean algebra describes switching circuits.

AL

Ada Lovelace

1815–1852 · English · mathematician, London

Notes on the Analytical Engine · 1843

She translated an article about Babbage's Analytical Engine and in 1843 added long notes to it. The notes included a step-by-step method for calculating Bernoulli numbers.

first algorithm published for a machine | Note G | symbol manipulation

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking** **Chapter 4.3 · Introduction to Programming**

Why it matters: Note G is often regarded as the first computer program. Lovelace realised that such a machine could process not only numbers but also symbols.

She was the daughter of the poet Lord Byron. The Ada programming language (1980) carries her name. Ada Lovelace Day celebrates women in science and technology.

ÉL

Édouard Lucas

1842–1891 · French · mathematician, Paris

Tower of Hanoi puzzle · 1883

He invented the Tower of Hanoi puzzle and put it on the market in 1883 under the pseudonym "N. Claus de Siam".

Tower of Hanoi | $2^n - 1$ moves | recursive thinking

CHAPTERS **Chapter 12 · Recursion**

Why it matters: Moving n disks reduces to moving n − 1 disks twice. It is the textbook example of a problem whose recursive solution is far simpler than its loop solution.

Lucas also studied the Fibonacci numbers and gave them that name. The Lucas numbers and Lucas primality tests are also named after him.

HH

Herman Hollerith

1860–1929 · American · Tabulating Machine Company

Punched-card tabulator · 1890 US census

He built electromechanical machines that counted data recorded on punched cards. These machines processed the data of the 1890 US census.

data processing | punched cards | tabulating machines

CHAPTERS **Chapter 1.1 · Basic Concepts**

Why it matters: Tabulating machines turned data processing into an industry: input, processing and output were now done by machines.

The Tabulating Machine Company he founded (1896) merged into the Computing-Tabulating-Recording Company in 1911. That company was renamed International Business Machines (IBM) in 1924.

AC

Alonzo Church

1903–1995 · American · Princeton University, UCLA

Lambda calculus · Church–Turing thesis

He developed the lambda calculus. In 1936 he used it to show that the decision problem of logic (the Entscheidungsproblem) has no general solution.

lambda calculus | computability | undecidability

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking**

Why it matters: According to the Church–Turing thesis, anything that can be computed by an effective method can also be computed by a Turing machine. The lambda calculus has the same computing power as the Turing machine.

Church reached the result shortly before Turing and by a different route. He supervised Turing's doctorate at Princeton (1938). The lambda calculus is the foundation of functional programming and of lambda expressions in today's languages.

JvN

John von Neumann

1903–1957 · Hungarian-born American · Institute for Advanced Study, Princeton

Stored-program architecture · 1945 · merge sort

He described the stored-program computer, in which program and data sit in the same memory. The description is in his 1945 *First Draft of a Report on the EDVAC*.

von Neumann architecture | fetch–decode–execute | merge sort

CHAPTERS **Chapter 1.3 · Computers**

Chapter 1.5 · Computation and Computational Thinking **Chapter 4.1 · Algorithms**

Chapter 4.2 · Flowcharts **Chapter 12 · Recursion**

Why it matters: The processor–memory arrangement discussed in the hardware chapters, and the "von Neumann bottleneck", carry his name.

Merge sort (1945) is attributed to him. With Herman Goldstine he introduced flowcharts for planning programs (1947). He also made major contributions to game theory, quantum mechanics and numerical analysis.

TF

Tommy Flowers

1905–1998 · English · Post Office Research Station, Dollis Hill

Colossus · 1944

He designed and built Colossus, an electronic machine that ran on vacuum tubes. From early 1944, Colossus helped break German Lorenz ciphers at Bletchley Park.

vacuum tubes | electronic computation | codebreaking

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking**

Why it matters: Colossus showed that large numbers of tubes could work reliably in digital computation. It was programmed with switches and plugs but did not store its program in memory.

Colossus Mark 1 had about 1,600 tubes and Mark 2 about 2,400. The machines were kept secret for decades, so Flowers's work was little known until the 1970s.

GH

Grace Hopper

1906–1992 · American · US Navy (rear admiral), Remington Rand

A-0 compiler · FLOW-MATIC · COBOL

In 1952 she developed the A-0 system, one of the first compilers. She also created the English-like language FLOW-MATIC, which strongly influenced COBOL (1959).

compilers | COBOL | English-like syntax

CHAPTERS **Chapter 4.3 · Introduction to Programming**

Why it matters: She argued that programs should be written in languages people can read and translated automatically. This idea lies behind every compiler in the course.

During the Second World War she programmed the Harvard Mark I. The moth her team taped into the logbook in 1947 is a famous story about computer "bugs"; the word had in fact been used earlier. In 2016 she was posthumously awarded the Presidential Medal of Freedom.

ME

John Mauchly & J. Presper Eckert

1907–1980 · 1919–1995 · American · Moore School, University of Pennsylvania

ENIAC · 1945–46 · UNIVAC I

They led the design of ENIAC, a general-purpose electronic computer. ENIAC was completed in 1945 and unveiled in February 1946.

ENIAC | vacuum tubes | UNIVAC

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking**

Why it matters: ENIAC showed that large-scale electronic computation was possible. The difficulty of reprogramming it with cables led directly to the stored-program design.

ENIAC had about 18,000 vacuum tubes. The machine was programmed by setting switches and cables. The company they founded built the UNIVAC I; the first machine was delivered to the US Census Bureau in 1951.

BBS

John Bardeen, Walter Brattain & William Shockley

1908–1991 · 1902–1987 · 1910–1989 · American · Bell Labs

Transistor · 1947

Bardeen and Brattain built the first point-contact transistor at Bell Labs in December 1947. Shortly afterwards Shockley designed the junction transistor.

Awards: 1956 Nobel Prize in Physics (all three together); Bardeen won a second Nobel Prize in Physics in 1972

transistor | semiconductors | switches

CHAPTERS **Chapter 1.1 · Basic Concepts** **Chapter 1.3 · Computers**

Why it matters: A transistor is a tiny electronic switch. Logic gates and processors are built from millions, even billions, of transistors.

Transistors replaced vacuum tubes because they were smaller, faster, cooler and more reliable. The company Shockley later founded contributed to the birth of Silicon Valley. Some engineers who left that company founded Fairchild Semiconductor.

KZ

Konrad Zuse

1910–1995 · German · civil engineer, Berlin

Z3 · 1941 · Plankalkül

In 1941 he completed the Z3, a programmable, fully automatic digital computer built from relays. He also designed Plankalkül, one of the first high-level programming languages.

binary floating point | relays | Plankalkül

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking** **Chapter 4.3 · Introduction to Programming**

Why it matters: The Z3 used binary arithmetic and floating-point numbers. These ideas are central to data representation and to Java's `double` type.

The Z3 read its program from punched film and was destroyed in an air raid in 1943. Plankalkül was designed in 1942–1945 but was published only in 1972.

AT

Alan Turing

1912–1954 · English · University of Cambridge, Bletchley Park, University of Manchester

Turing machine · 1936

He defined the Turing machine in his paper "On Computable Numbers" (1936). He proved that some problems cannot be solved by any algorithm.

Turing machine | universal machine | computability

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking**

Why it matters: The universal Turing machine reads the rules of another machine from its own tape. This is the theoretical form of the stored-program idea.

In the Second World War he led the codebreaking work on the Enigma cipher at Bletchley Park. He designed the ACE computer (1945) and proposed the Turing test (1950). The ACM Turing Award, first given in 1966, carries his name.

HG

Herman Goldstine

1913–2004 · American · ENIAC project, Institute for Advanced Study

Flow diagrams · 1947

With von Neumann he wrote the reports "Planning and Coding of Problems for an Electronic Computing Instrument" (1947–48). These reports introduced flowcharts for program design.

flowcharts | program planning | IAS machine

CHAPTERS **Chapter 4.3 · Introduction to Programming** **Chapter 4.2 · Flowcharts**

Why it matters: The boxes and arrows of the flowcharts used in the algorithms chapters come from these reports.

As an army officer he was the liaison to the ENIAC project, and in 1944 he brought von Neumann into it. He later directed the IAS computer project. He also wrote a history of computing, *The Computer from Pascal to von Neumann* (1972).

CS

Claude Shannon

1916–2001 · American · MIT, Bell Labs

Switching circuits · 1937 · information theory · 1948

In his 1937 master's thesis he showed that Boolean algebra can describe and simplify relay switching circuits. In 1948 he founded information theory.

logic circuits | bit | information theory

CHAPTERS **Chapter 1.1 · Basic Concepts** **Chapter 1.3 · Computers**

Why it matters: His thesis connects Boolean logic to the logic gates in processors. His 1948 paper made the bit the unit of information.

The word "bit" (binary digit) appears in the 1948 paper; Shannon credits it there to John Tukey. Data compression and reliable communication over networks rest on information theory.

BB

Bob Bemer

1920–2004 · American · IBM and other companies

ASCII · escape sequence · backslash

He played a leading role in creating ASCII, first published in 1963. The escape sequence and the backslash (\) character are also among his contributions.

ASCII | character codes | escape sequences

CHAPTERS **Chapter 1.1 · Basic Concepts** **Chapter 1.3 · Computers**

Why it matters: ASCII's 7-bit codes (A = 65) are the first 128 code points of Unicode. The character arithmetic used in the course is based on these codes.

ASCII was a committee effort, but Bemer is often called its father. He proposed the backslash in 1961. He was also one of the first to draw attention to the year 2000 problem.

BJ

Corrado Böhm & Giuseppe Jacopini

1923–2017 (Böhm) · Italian · Rome (IAC-CNR); Jacopini was Böhm's student

Structured program theorem · 1966

They proved that any program written as a flowchart can be rewritten using only sequence, selection and repetition (Communications of the ACM, 1966).

sequence | selection | repetition

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking** **Chapter 4.2 · Flowcharts** **Chapter 7 · Conditionals** **Chapter 8 · Loops**

Why it matters: The three control structures (sequence, selection and loops) are enough for every algorithm; `goto`-style jumps are not needed.

This result formed the theoretical basis of structured programming, which Dijkstra championed from 1968 on. Böhm later also did important work on the lambda calculus.

KN

Jack Kilby & Robert Noyce

1923–2005 · 1927–1990 · American · Texas Instruments (Kilby); Fairchild Semiconductor, Intel (Noyce)

Integrated circuit · 1958–59

They invented the integrated circuit independently of each other. Kilby demonstrated the first example at Texas Instruments in 1958. In 1959 Noyce developed the practical planar silicon form at Fairchild.

Awards: Kilby: 2000 Nobel Prize in Physics (Noyce had died in 1990)

integrated circuit · chips · miniaturisation

CHAPTERS **Chapter 1.2 · Computer Hardware** **Chapter 1.3 · Computers**

Why it matters: Fitting many transistors on a single chip made processors, memory and today's computers possible.

Noyce co-founded Intel in 1968 with Gordon Moore. Kilby's team also developed the first handheld electronic calculator (1967).

JB

John Backus

1924–2007 · American · IBM

FORTRAN · 1957 · Backus–Naur Form

He led the team at IBM that developed FORTRAN (1957). With its optimising compiler, FORTRAN was the first widely used high-level language.

Awards: 1977 Turing Award

FORTRAN · compilers · BNF grammar

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking** **Chapter 4.3 · Introduction to Programming**

Why it matters: FORTRAN showed that compiled code can run at nearly the speed of hand-written machine code. This made high-level languages practical.

He proposed a notation for writing the syntax of languages. Peter Naur used it in the ALGOL 60 report. The notation, known as Backus–Naur Form (BNF), is still used today to define language grammars.

DE

Douglas Engelbart

1925–2013 · American · Stanford Research Institute (SRI)

Computer mouse · "Mother of All Demos", 1968

He invented the computer mouse; he built the first one with Bill English in the mid-1960s. In 1968 he demonstrated windows, hypertext and real-time collaborative document editing live.

Awards: 1997 Turing Award

mouse · hypertext · interactive computing

CHAPTERS **Chapter 1.2 · Computer Hardware** · **Chapter 2.1 · Software and Operating Systems** · **Chapter 2.2 · The Windows Operating System**

Why it matters: The pointing and window ideas of graphical desktops such as Windows are based on his work.

His demonstration in San Francisco in December 1968 later became known as the "Mother of All Demos". His goal was to use computers to augment human intellect.

BD

Paul Baran & Donald Davies

1926–2011 · 1924–2000 · Polish-born American · RAND (Baran); Welsh · National Physical Laboratory (Davies)

Packet switching · 1960s

They developed packet switching independently of each other. In this method a message is split into small blocks. The blocks travel through the network separately and are reassembled at the destination.

packet switching · distributed networks · routing

CHAPTERS **Chapter 1.4 · The Internet**

Why it matters: Every message on the Internet is carried as independent packets.

Baran described distributed networks in a series of RAND reports in 1964. Davies coined the term "packet" and built a packet-switched network at the NPL in Britain. The work of both influenced the design of ARPANET (1969).

KK

John Kemeny & Thomas Kurtz

1926–1992 · 1928–2024 · Hungarian-born American; American · Dartmouth College

BASIC · 1964

They developed the BASIC language and the Dartmouth Time-Sharing System. The first BASIC program ran on this system on 1 May 1964.

BASIC | time sharing | programming for everyone

CHAPTERS **Chapter 4.3 · Introduction to Programming**

Why it matters: BASIC was designed so that students outside the sciences could also learn programming. It became the first language of many home-computer users.

Microsoft's first product was a BASIC interpreter (1975). Kemeny was president of Dartmouth from 1970 to 1981. Together with Kurtz he later founded the company True BASIC.

JM

John McCarthy

1927–2011 · American · MIT, Stanford University

Lisp · 1958 · "artificial intelligence"

He designed Lisp, a language built on recursive functions (1958). He coined the term "artificial intelligence" in the 1955 Dartmouth proposal.

Awards: 1971 Turing Award

recursion | conditional expressions | Lisp

CHAPTERS **Chapter 4.3 · Introduction to Programming** **Chapter 7 · Conditionals** **Chapter 12 · Recursion**

Why it matters: Lisp made recursion an ordinary programming tool. The language also introduced the conditional expression, the ancestor of the `if-else` construct.

He described Lisp in the paper "Recursive Functions of Symbolic Expressions and Their Computation by Machine" (1960). He also invented garbage collection and proposed time sharing.

SP

Seymour Papert

1928–2016 · South African-born · MIT

Logo · *Mindstorms* (1980)

He co-developed the Logo language for children (1967). He used the phrase "computational thinking" in his book *Mindstorms* (1980).

Logo | learning by programming | computational thinking

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking**

Why it matters: He argued that programming is a way of thinking about problems. Jeannette Wing later developed this idea.

In Logo's turtle graphics the student gives commands step by step and sees the result immediately. The LEGO Mindstorms robot kits take their name from his book.

GM

Gordon Moore

1929–2023 · American · Fairchild Semiconductor, Intel

Moore's law · 1965

In 1965 he observed that the number of components on a chip was doubling every year. In 1975 he revised the rate to about once every two years.

Moore's law | transistor count | semiconductors

CHAPTERS **Chapter 1.1 · Basic Concepts** **Chapter 1.2 · Computer Hardware**

Why it matters: Moore's law explains the steady growth in processor power and memory capacity discussed in the hardware presentation.

He co-founded Intel in 1968 with Robert Noyce. The trend lasted for decades. As transistors approached physical limits, the growth slowed.

ED

Edsger W. Dijkstra

1930–2002 · Dutch · Mathematisch Centrum, Eindhoven University of Technology, University of Texas at Austin

Structured programming · "Go To Statement Considered Harmful", 1968

In 1968 he argued that unrestricted `goto` jumps make programs incomprehensible. This article started the structured programming movement.

Awards: 1972 Turing Award

structured programming | shortest paths | semaphores

CHAPTERS **Chapter 2.1 · Software and Operating Systems**

Chapter 4.3 · Introduction to Programming **Chapter 4.2 · Flowcharts** **Chapter 8 · Loops**

Why it matters: Writing programs with a clear sequence, selection and loops, as in Java, follows his ideas.

He published his shortest-path algorithm in 1959. He introduced semaphores to coordinate processes. The "dining philosophers" problem illustrates deadlock, one of the topics of the operating systems presentation.

WK

William Kahan

b. 1933 · Canadian · University of California, Berkeley

IEEE 754 floating point · 1985

He was the chief architect of the IEEE 754 standard (1985) for floating-point arithmetic.

Awards: 1989 Turing Award

floating point | rounding | double

CHAPTERS **Chapter 1.3 · Computers** **Chapter 6 · Variables, Data Types and Operators**

Why it matters: Java's `float` and `double` types follow IEEE 754. This is why `0.1 + 0.2` prints `0.30000000000000004`.

The standard defines formats made of sign, exponent and fraction bits, the rounding rules, and special values such as infinity and NaN. As a result, the same program gives the same result on different machines.

TH

Tony Hoare

1934–2026 · English · Elliott Brothers, Queen's University Belfast, University of Oxford, Microsoft Research

Quicksort · 1959–61

He devised quicksort around 1959–60 and published it in 1961. The algorithm partitions the array into two parts around a pivot and then sorts the parts recursively.

Awards: 1980 Turing Award

quicksort | divide and conquer | Hoare logic

CHAPTERS **Chapter 4.1 · Algorithms** **Chapter 12 · Recursion**

Why it matters: Like merge sort, quicksort is a divide-and-conquer algorithm. Its average running time is O(n log n); on large arrays it is much faster than bubble sort.

He developed Hoare logic (1969) for proving programs correct and CSP for concurrency. He added the null reference to ALGOL W (1965), which he later called his "billion-dollar mistake". He died on 5 March 2026.

NW

Niklaus Wirth

1934–2024 · Swiss · ETH Zurich

Pascal · 1970 · *Algorithms + Data Structures = Programs*

He designed Pascal (1970) to teach structured programming. He then developed the languages Modula-2 and Oberon.

Awards: 1984 Turing Award

Pascal | structured programming | stepwise refinement

CHAPTERS **Chapter 4.3 · Introduction to Programming**

Why it matters: Pascal made structured, strongly typed programming the standard of introductory courses for twenty years.

The title of his book *Algorithms + Data Structures = Programs* (1976) sums up one of the main ideas of this course. He also advocated stepwise refinement: a problem is solved by dividing it into smaller steps.

EC

Edward G. Coffman Jr.

b. 1934 · American · Pennsylvania State University, Bell Laboratories, Columbia University

The four conditions for deadlock · 1971

In the paper "System Deadlocks" (*Computing Surveys*, 1971), written with M. J. Elphick and Arie Shoshani, he defined the four conditions that must hold at the same time for a deadlock to occur. These are known today as the Coffman conditions.

deadlock | Coffman conditions | resource management

CHAPTERS **Chapter 2.1 · Software and Operating Systems**

Why it matters: A deadlock cannot occur unless mutual exclusion, hold and wait, no preemption and circular wait all hold together; breaking any one is enough. A process that holds the printer and waits for the file, and another that holds the file and waits for the

printer, is the simplest example of circular wait. If all processes request resources in the same order, this cycle cannot form.

In 1973 he wrote the book *Operating Systems Theory* with Peter J. Denning. He worked on scheduling, bin packing and queueing theory, and is among the founders of the ACM Symposium on Operating Systems Principles.

HFS

Ted Hoff, Federico Faggin & Masatoshi Shima

b. 1937 · b. 1941 · b. 1943 · American; Italian-born American; Japanese · Intel and Busicom

Intel 4004 microprocessor · 1971

Together with Stanley Mazor, they developed the Intel 4004 (1971), the first commercial single-chip microprocessor.

microprocessor | CPU on a chip | 4-bit

CHAPTERS **Chapter 1.2 · Computer Hardware** **Chapter 1.3 · Computers**

Why it matters: Fitting the entire central processing unit (CPU) on one chip opened the way to personal computers, phones and today's processors.

Hoff proposed the architecture; Shima of Busicom worked on the logic design; Faggin led the silicon design. The 4004 had about 2,300 transistors. Faggin and Shima later designed the Zilog Z80.

DK

Donald Knuth

b. 1938 · American · Stanford University

***The Art of Computer Programming* (since 1968)**

He turned the analysis of algorithms into a systematic field in the series *The Art of Computer Programming*. The first volume appeared in 1968.

Awards: 1974 Turing Award

algorithm analysis | big O | sorting and searching

CHAPTERS **Chapter 4.3 · Introduction to Programming** **Chapter 4.1 · Algorithms**

Why it matters: Comparing linear and binary search, or bubble sort and merge sort, by counting steps is the method he made standard.

Volume 3 of the series treats sorting and searching in detail. He also created the TeX typesetting system (1978) and advocated "literate programming".

AK

Alan Kay

b. 1940 · American · Xerox PARC

Smalltalk · overlapping windows · Dynabook

In the 1970s at Xerox PARC he led the team that developed the Smalltalk language and the graphical interface with overlapping windows.

Awards: 2003 Turing Award

graphical user interface | object-oriented programming | Smalltalk

CHAPTERS **Chapter 2.1 · Software and Operating Systems** **Chapter 2.2 · The Windows Operating System**

Why it matters: The desktop with windows, icons and menus, later popularised by Apple and Microsoft, grew out of this work.

The term "object-oriented" is attributed to him for the approach used in Smalltalk; Smalltalk also influenced Java. The Dynabook concept (1972) envisioned a portable, tablet-like computer for children.

RT

Dennis Ritchie & Ken Thompson

1941–2011 · b. 1943 · American · Bell Labs

Unix · 1969 · C language · 1972

They developed the Unix operating system at Bell Labs starting in 1969. Ritchie developed the C language around 1972, and Unix was rewritten in it.

Awards: 1983 Turing Award (shared)

Unix | C language | portability

CHAPTERS **Chapter 1.3 · Computers** **Chapter 2.1 · Software and Operating Systems** **Chapter 4.3 · Introduction to Programming**

Why it matters: Linux, macOS and Android follow Unix ideas. Java's syntax (curly braces, `if`, `for`, `while`) also comes from C.

Thompson also created the B language and, with Rob Pike, designed the UTF-8 encoding (1992). At Google he became one of the designers of the Go language (2009).

RT

Ray Tomlinson

1941–2016 · American · BBN (Bolt Beranek and Newman)

Network email · 1971 · @ sign

In 1971 he sent the first email between computers on ARPANET. He chose the @ sign to separate the user name from the computer name.

email | @ sign | ARPANET

CHAPTERS **Chapter 1.4 · The Internet**

Why it matters: The user@host form is used in every email address today.

In earlier systems only users of the same computer could leave messages for each other. Tomlinson carried this over a network. In 2012 he was inducted into the Internet Hall of Fame.

BK

Brian Kernighan

b. 1942 · Canadian · Bell Labs, Princeton University

"hello, world" · *The C Programming Language* (1978)

He wrote the earliest known "hello, world" programs in Bell Labs's tutorial documents for B (1972) and C (1974). The program became famous through the book he wrote with Dennis Ritchie (1978).

hello, world | first program | programming style

CHAPTERS **Chapter 4.3 · Introduction to Programming**

Chapter 5 · Getting Started with Java

Why it matters: Starting a new language by printing a short greeting is a tradition he began. The first "Hello" program in Java continues that tradition.

He is the "K" in the name of the AWK language. The books he co-wrote on programming style were very influential. *The C Programming Language* is often called "K&R".

CK

Vint Cerf & Bob Kahn

b. 1943 · b. 1938 · American · Stanford, DARPA

TCP/IP · 1974

They designed the TCP/IP protocols, which connect different networks to one another (the paper was published in 1974). ARPANET switched to TCP/IP on 1 January 1983.

Awards: 2004 Turing Award (shared); 2005 US Presidential Medal of Freedom

TCP/IP | internetworking | protocols

CHAPTERS **Chapter 1.4 · The Internet**

Why it matters: As the layered-protocol slides show, TCP/IP is the "common set of rules" that joins networks into the Internet.

IP carries packets between networks using addresses. TCP delivers the data reliably and in order. Both are often called the "fathers of the Internet".

GA

Robert Gaskins & Dennis Austin

b. 1943 · 1947–2023 · American · Forethought, Inc., later Microsoft

PowerPoint · 1987

Gaskins designed the presentation program called Presenter, and Austin led its programming. It was released in 1987 as PowerPoint 1.0 for the Macintosh.

presentation software | slides | Forethought

CHAPTERS **Chapter 3.3 · Microsoft PowerPoint**

Why it matters: With PowerPoint, slide-based presentation software became a standard office tool.

Microsoft bought Forethought in 1987 for about 14 million dollars, one of its first acquisitions. PowerPoint later became part of Microsoft Office.

CS

Charles Simonyi

b. 1948 · Hungarian-born American · Xerox PARC, Microsoft

Bravo editor · Microsoft Word

At Xerox PARC he developed Bravo with Butler Lampson. Bravo was one of the first editors in which what you see on screen is exactly what gets printed (WYSIWYG, "what you see is what you get"). Simonyi later led the development of Microsoft Word (1983).

word processor | WYSIWYG | Microsoft Office

CHAPTERS **Chapter 3.1 · Microsoft Word**

Why it matters: Editing a document on screen in the form in which it will be printed is the basic idea of Word. The idea comes from Bravo.

He joined Microsoft in 1981 and led the applications group. Multiplan and Excel were also products of this group. "Hungarian notation" in variable naming is named in reference to him. He travelled to the International Space Station twice (2007, 2009).

PM

Paul Mockapetris

b. 1948 · American · USC Information Sciences Institute

Domain Name System (DNS) · 1983

He designed the Domain Name System (DNS), which was defined in RFC 882 and 883 (November 1983). DNS translates names such as www.example.edu into IP addresses.

DNS | domain names | hierarchy

CHAPTERS **Chapter 1.4 · The Internet**

Why it matters: The step-by-step DNS resolution (root server, top-level domain, authoritative server) follows his design.

DNS replaced a single HOSTS.TXT file that listed every computer on the network. It is a distributed, hierarchical database that uses caching. The RFCs were updated in 1987 by RFC 1034 and 1035.

BS

Bjarne Stroustrup

b. 1950 · Danish · Bell Labs, Texas A&M, Columbia University

C++ · 1983

He created the C++ language. It began in 1979 as "C with Classes", was named C++ in 1983, and was released commercially in 1985.

C++ | object-oriented programming | classes

CHAPTERS **Chapter 4.3 · Introduction to Programming**

Why it matters: C++ brought classes and objects into C-style syntax. Java followed this approach in a simpler and safer form.

The name uses C's increment operator `++`, the same operator taught in Java loops. C++ is widely used in system software, games and software where performance matters.

BF

Dan Bricklin & Bob Frankston

b. 1951 · b. 1949 · American · Software Arts

VisiCalc · 1979

They developed VisiCalc, the first spreadsheet program for personal computers, for the Apple II in 1979.

spreadsheets | cells and formulas | killer app

CHAPTERS **Chapter 3.2 · Microsoft Excel**

Why it matters: The foundations of Excel (rows, columns, cell references and automatic recalculation) began with VisiCalc.

The idea came to Bricklin while he was a Harvard Business School student; Frankston wrote most of the code. Many people bought a personal computer just to run VisiCalc, which is why it is considered the first "killer app".

RS

Richard Stallman

b. 1953 · American · MIT AI Lab, Free Software Foundation

GNU project · 1983 · GPL

In 1983 he announced the GNU project to build a free, Unix-like operating system. In 1985 he founded the Free Software Foundation.

free software | GNU | GPL licence

CHAPTERS **Chapter 1.1 · Basic Concepts** **Chapter 2.1 · Software and Operating Systems**

Why it matters: Software licences and open source rest on the free software movement he started.

The GNU General Public License (GPL, 1989) grants the right to use, study, modify and share software. The only condition is that modified versions are distributed under the same terms. Linux systems combine the Linux kernel with GNU tools.

GA

Bill Gates & Paul Allen

b. 1955 · 1953–2018 · American · Microsoft

Microsoft · 1975 · MS-DOS · Windows

They founded Microsoft in 1975. They supplied MS-DOS for the IBM PC (1981) and released Windows 1.0 in 1985.

MS-DOS | Windows | Microsoft Office

CHAPTERS **Chapter 2.1 · Software and Operating Systems**

Chapter 2.2 · The Windows Operating System

Why it matters: Windows and Microsoft Office, used in the course, are products of the company they founded.

Their first product was a BASIC interpreter for the Altair 8800. MS-DOS was based on 86-DOS, bought from Seattle Computer Products. The Office suite was first offered as a bundle around 1990.

TBL

Tim Berners-Lee

b. 1955 · English · CERN, MIT, University of Oxford

World Wide Web · 1989

He proposed the World Wide Web (the Web) at CERN in 1989. He developed the first web browser, the first web server, HTML and HTTP. The first website went live in 1991.

Awards: 2016 Turing Award

WWW | HTML | HTTP and URLs

CHAPTERS **Chapter 1.4 · The Internet**

Why it matters: The Web is a service that runs on top of the Internet. URLs, HTTP and browsers are his design.

In 1993 CERN made the web technology free for everyone to use, which let the Web spread quickly. Berners-Lee founded the World Wide Web Consortium (W3C) in 1994.

JG

James Gosling

b. 1955 · Canadian · Sun Microsystems

Java · 1995

He led the design of the Java language at Sun Microsystems. Java was released in 1995 with the slogan "write once, run anywhere".

Java | bytecode and JVM | platform independence

CHAPTERS **Chapter 4.3 · Introduction to Programming**

Chapter 5 · Getting Started with Java **Chapter 6 · Variables, Data Types and Operators**

Why it matters: The programming part of the course uses Java. Source code is compiled with `javac` into bytecode (an intermediate code); the Java Virtual Machine (JVM) runs this code on every platform.

The language was born in 1991 under the name Oak, in Sun's Green project aimed at consumer devices. Oracle bought Sun in 2010 and maintains Java today.

GvR

Guido van Rossum

b. 1956 · Dutch · CWI Amsterdam, later Google, Dropbox, Microsoft

Python · 1991

He created Python, a language that puts a premium on readable code. The first version came out in February 1991.

Python | interpreter | readability

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking** **Chapter 4.3 · Introduction to Programming** **Chapter 4.1 · Algorithms**

Why it matters: In the course, Python is the example of an interpreted language (one that uses bytecode and a virtual machine). It is compared with compiled C and hybrid Java.

Python marks blocks by indentation rather than curly braces. Van Rossum led the language until 2018 with the title "Benevolent Dictator For Life".

JW

Jeannette Wing

b. 1956 · American · Carnegie Mellon University, Columbia University

"Computational Thinking" · 2006

In her essay published in Communications of the ACM in 2006, she argued that computational thinking is a fundamental skill for everyone, not just for computer scientists.

decomposition | abstraction | algorithmic thinking

CHAPTERS **Chapter 1.5 · Computation and Computational Thinking**

Why it matters: Her essay is one of the sources of the computational thinking presentation. The four components in that presentation are decomposition, pattern recognition, abstraction and algorithm design.

She also worked on formal methods. The Liskov–Wing substitution principle (1994) is a rule of object-oriented design. She later headed the computer science directorate at the US National Science Foundation (NSF) and was a corporate vice president at Microsoft

Research.

LT

Linus Torvalds

b. 1969 · Finnish-born American · University of Helsinki, Linux Foundation

Linux kernel · 1991 · Git

He started the Linux kernel in 1991 as a student in Helsinki and released it as open-source software.

Linux kernel | open source | Git

CHAPTERS **Chapter 1.1 · Basic Concepts** **Chapter 2.1 · Software and Operating Systems**

Why it matters: The Linux kernel runs underneath Android, most web servers and supercomputers. When operating systems are discussed, the difference between a kernel and a full distribution is explained through this example.

He announced the project on 25 August 1991, calling it "just a hobby". In 2005 he created Git, today the most widely used version control system in software development.

UC

Joe Becker, Lee Collins & Mark Davis

American · Xerox and Apple · Unicode Consortium

Unicode · 1987–91

They started the Unicode project in 1987 to give every character in every writing system a single code. The design was laid out in Becker's 1988 paper "Unicode 88".

Unicode | code points | UTF-8

CHAPTERS **Chapter 1.1 · Basic Concepts** **Chapter 1.3 · Computers** **Chapter 6 · Variables, Data Types and Operators** **Chapter 11 · Strings**

Why it matters: Java's `char` and `String` types are based on Unicode. That is why the code of the character 'A' is 65, and why letters outside ASCII, such as é (U+00E9), can be stored and compared.

Becker and Collins worked at Xerox, Davis at Apple. The Unicode Consortium was founded in 1991 and Unicode 1.0 appeared the same year. UTF-8 (1992), designed by Ken Thompson and Rob Pike, became the most widely used encoding.

Award years show the year the prize was given; the Turing Award is given by the ACM (Association for Computing Machinery). People whose work is described together share a card. On cards without a birth year, the order follows the period of the main work. On the cards, “b.” means the year of birth and “c.” means approximately.