Chapter 2.2: Syntax Rules

Object Oriented Programming with Java

.java files and compilation | Bytecode and the JVM | Variables | Primitive data types | Identifiers and keywords | Assignment and constants | Type conversion | Operators | The ternary operator ?: | Control statement syntax

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/java/slides/Chapter_02_02_Syntax_Rules.html

Learning Outcomes

Compiling and running

Explain how a `.java` file becomes bytecode with `javac` and a running program with `java`; follow the levels of abstraction and bytecode executing on the JVM stack; explain how the JVM makes Java platform independent.

Variables and types

Know the eight primitive types with their sizes and ranges; declare variables, assign values, and spot overflow and floating-point precision issues.

Rules and naming

Tell valid identifiers from invalid ones, recognize keywords, and follow the camelCase / PascalCase / UPPER_CASE conventions.

Expressions and flow

Evaluate expressions by operator precedence and convert between types; pick a value by condition with `?:`; recognize the syntax of the `if`, `switch`, `for`, `while`, `do-while`, `break`, `continue` statements.

Software Development with Java

- Java code is written in **plain text** files, not in formatted documents such as Word.
- Source code files have the extension `.java`; compilers and IDEs recognize files by it.
- If a file contains a `public` class, the file name must be the **same** as the class name: `HelloWorld` → `HelloWorld.java`.

Readable

Plain text is readable by people; code is easy to understand and share.

Portable

The same `.java` file opens on every operating system and in every IDE.

Easy to fix

Any text editor is enough to fix errors or update the code.

Case matters in the file name too: `helloworld.java` with `public class HelloWorld` does not compile.

Programming Tools

Tool	What it does	Examples
Text editor	Write source code as plain text	Notepad++, VS Code
JDK (Java Development Kit)	Compiler, launcher and other development tools	`javac`, `java`, `javap`, `jshell`
IDE (Integrated Development Environment)	Editor + compile + run + debug in one window; underlines errors as you type	Eclipse, IntelliJ IDEA, NetBeans, VS Code (with Java extensions)
JShell (Java 9+)	Try one-line Java expressions without writing a class	`jshell> 7 / 2` → `3`

Using `javac` and `java` on the command line at least once shows what happens behind the IDE's “Run” button.

A Sample .java File: HelloWorld.java

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

`public class HelloWorld`

Every Java program contains at least one **class**. Its body goes between the `{ }` braces.

`public static void main(String[] args)`

The program's **entry point**: the JVM starts running from this method. It must be written exactly like this.

`System.out.println(...)`

Prints the text in parentheses to the console and moves to the next line. Text goes in double quotes.

Semicolon `;`

Every statement ends with `;`. Blocks closed by a brace need no `;` after them.

Basic Syntax Rules

```
/* Multi-line comment:
   the compiler ignores it. */
public class Rules {
    // Single-line comment
    public static void main(String[] args) {
        int num = 5;  // ends with ;
        int Num = 7;  // num ≠ Num
        System.out.println(num
                + Num);   // a line can be split
    }
}
```

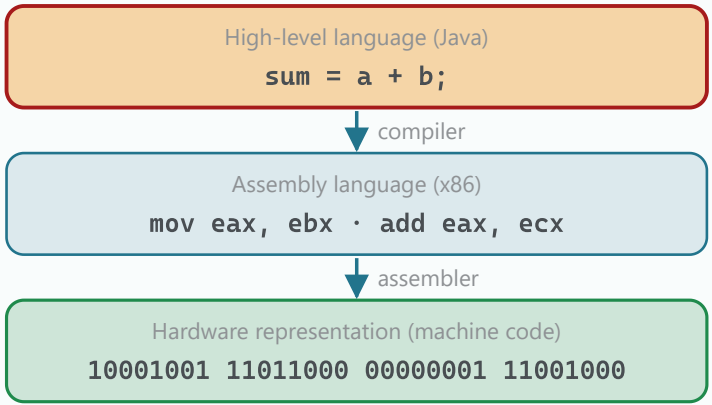
OUTPUT

```
12
```

- **Syntax**: the writing rules of a language. Code that breaks them **does not compile**.
- Every statement ends with `;`.
- Blocks open with `{` and close with `}`; every opened block must be closed.
- Java is case-sensitive: `num` ≠ `Num`, `System` ≠ `system`.
- Spaces, tabs and line breaks don't matter to the compiler; indentation is for readability.
- Comments: `//` runs to the end of the line, `/* … */` spans several lines. The compiler ignores both.

Levels of Abstraction in Programming

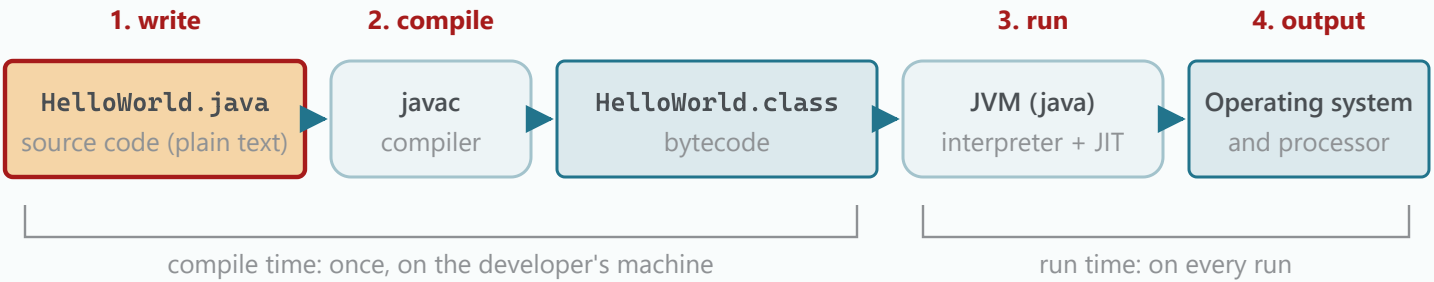
- **High-level language**: close to the problem domain; makes software more **productive**, readable and understandable. Java, C, Python...
- **Assembly language**: expresses instructions as text (`ADD` , `MOV`) that are translated into the hardware representation. A lower level of abstraction.
- The higher the abstraction, the **shorter and more readable the code**; a compiler or assembler translates it to the level below.
- **Hardware representation**: only **bits**. Instructions and data are understood directly by the hardware.



In the example, `a` , `b` , `sum` are assumed to be in registers `ebx` , `ecx` , `eax` , respectively; the machine code is the real x86 encoding (89 D8, 01 C8).

Compilation

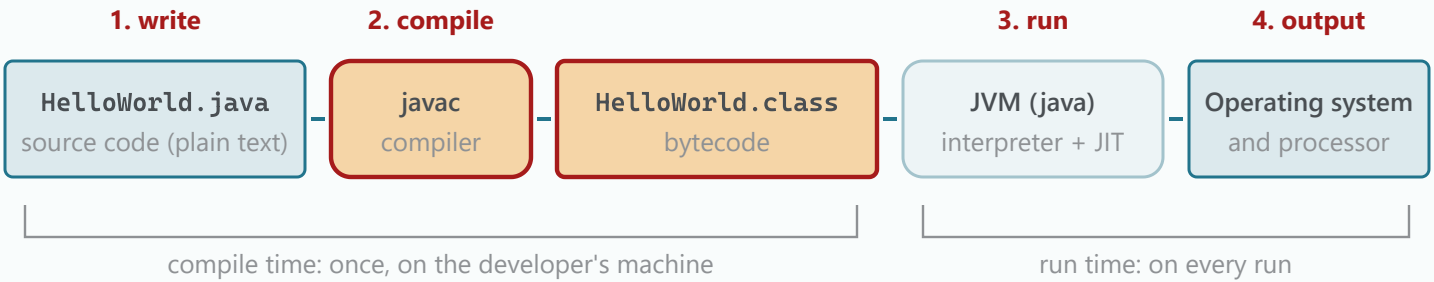
- A computer cannot run `.java` text directly; it must first be **compiled**. The compile command: `javac`.
- The Java compiler translates source code not into the processor's machine language but into **bytecode**, which the JVM understands, and writes it to a `.class` file. The JVM turns bytecode into machine code as it runs.



The programmer writes the source as plain text and saves it as `HelloWorld.java`.

Compilation

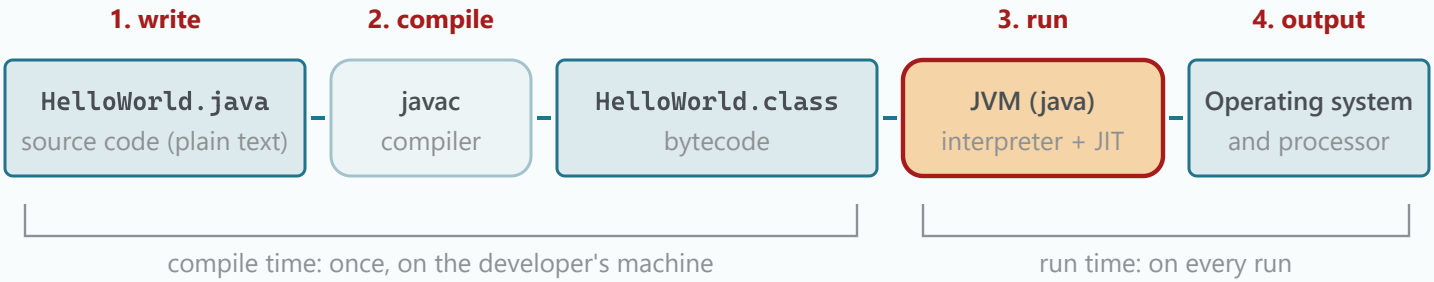
- A computer cannot run `.java` text directly; it must first be **compiled**. The compile command: `javac`.
- The Java compiler translates source code not into the processor's machine language but into **bytecode**, which the JVM understands, and writes it to a `.class` file. The JVM turns bytecode into machine code as it runs.



`javac HelloWorld.java`: the compiler checks syntax and types; if there are no errors, `HelloWorld.class` is created.

Compilation

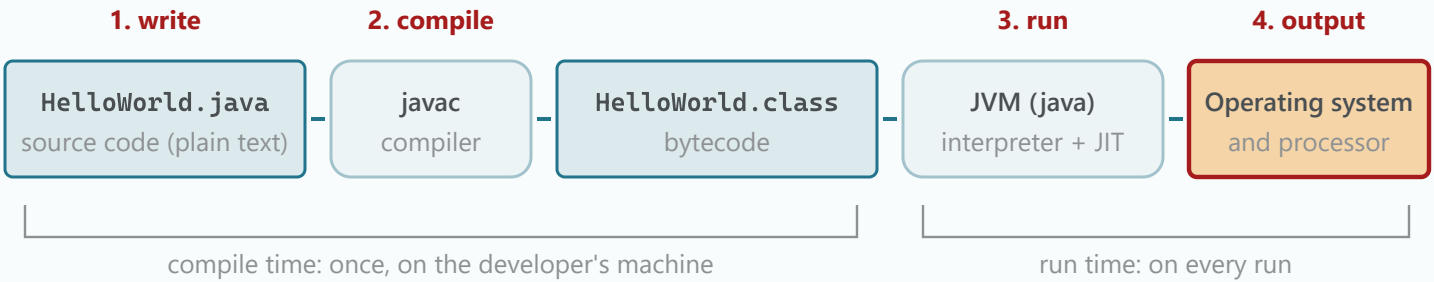
- A computer cannot run `.java` text directly; it must first be **compiled**. The compile command: `javac`.
- The Java compiler translates source code not into the processor's machine language but into **bytecode**, which the JVM understands, and writes it to a `.class` file. The JVM turns bytecode into machine code as it runs.



`java HelloWorld`: the launcher starts the JVM and loads the class; the JVM interprets the bytecode and JIT-compiles hot code to machine code.

Compilation

- A computer cannot run `.java` text directly; it must first be **compiled**. The compile command: `javac`.
- The Java compiler translates source code not into the processor's machine language but into **bytecode**, which the JVM understands, and writes it to a `.class` file. The JVM turns bytecode into machine code as it runs.



The operating system and processor execute the actual instructions; `Hello, World!` appears on the screen.

Why Compile?

- `HelloWorld.java`, once compiled, produces `HelloWorld.class` in the same folder.
- The compiler catches errors **before** the program runs: a missing `;`, an unclosed parenthesis, a misspelled name, a type mismatch…
- If there are errors, no `.class` file is produced; the **file:line** part of the message shows where the error is.
- Compilation and the JVM's run-time optimizations (JIT) help programs run faster.

If the `;` on line 3 is missing:

```
System.out.println("Hello, World!")
```

```
HelloWorld.java:3: error: ';' expected
        System.out.println("Hello, World!")
                                           ^
1 error
```

COMPILING AND RUNNING

Basic Steps with Java

1. **Compile:** the `.java` file becomes a `.class` file.
   `javac HelloWorld.java`
2. **Run:** `java` takes the class name **without the extension**.
   `java HelloWorld`
3. **Output:** the program prints `Hello, World!`.

COMMAND LINE

```
> javac HelloWorld.java
> dir /b
HelloWorld.class
HelloWorld.java
> java HelloWorld
Hello, World!
```

`dir` (Windows) / `ls` (Linux, macOS) lists the files in the folder. If `javac` prints nothing, compilation succeeded.



Common mistake: `java HelloWorld.class`

Don't type the extension when running; the JVM expects a **class name**.

Java 11+ shortcut

Single-file programs can be run directly with `java HelloWorld.java`, compiled on the fly (no `.class` file is written).

Bytecode (Intermediate Code)

- `.class` files are **not machine code**.
- Instead they hold **bytecode**, which the Java virtual machine understands: bytecode is the JVM's “machine language”.
- Each instruction's opcode is a single byte (hence the name), optionally followed by operand bytes.
- Bytecode is not tied to a processor or operating system: **the same** `.class` file runs on Windows, Linux and macOS.

Analogy

Bytecode is like **sheet music** that reads the same everywhere; each computer's JVM is a musician playing it on its own instrument.

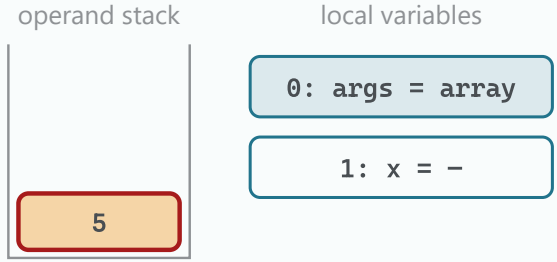
To see it

The JDK's `javap -c MyClass` command lists the bytecode instructions in a `.class` file in readable form.

Bytecode Example: Step by Step

`int x = 5; System.out.println(x);` compiled (`javap -c` , JDK 21). The JVM is a **stack machine**: instructions push values onto the operand stack and pop them off.

```
▶ 0: iconst_5
  1: istore_1
  2: getstatic     #7    // System.out
  5: iload_1
  6: invokevirtual #13   // println(int)
  9: return
```



`iconst_5` : pushes the constant 5 onto the operand stack. (Source: `int x = 5;`)

Step 1 / 6

VARIABLES

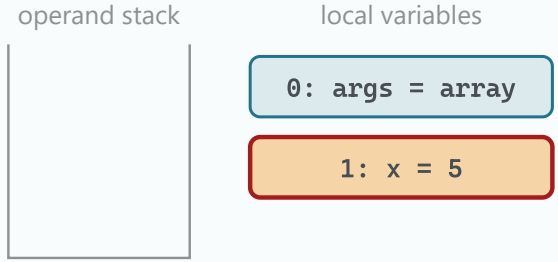
`x = –` `stack = []`

OUTPUT

Bytecode Example: Step by Step

`int x = 5; System.out.println(x);` compiled (`javap -c` , JDK 21). The JVM is a **stack machine**: instructions push values onto the operand stack and pop them off.

```
1 0: iconst_5
▶ 1: istore_1
3 2: getstatic     #7    // System.out
4 5: iload_1
5 6: invokevirtual #13   // println(int)
6 9: return
```



`istore_1` : pops the top of the stack into local variable 1 (`x`).

Step 2 / 6

VARIABLES

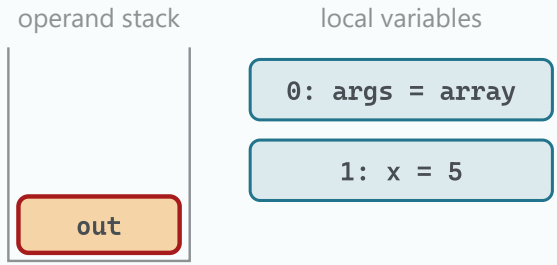
`x = 5` `stack = []`

OUTPUT

Bytecode Example: Step by Step

`int x = 5; System.out.println(x);` compiled (`javap -c` , JDK 21). The JVM is a **stack machine**: instructions push values onto the operand stack and pop them off.

```
1 0: iconst_5
2 1: istore_1
▶ 2: getstatic     #7    // System.out
4 5: iload_1
5 6: invokevirtual #13   // println(int)
6 9: return
```



`getstatic #7` : `System.out` is pushed onto the stack (as a reference). #7 is its constant-pool entry.

Step 3 / 6

VARIABLES

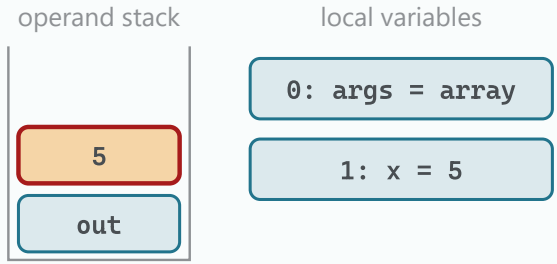
`x = 5` `stack = [System.out]`

OUTPUT

Bytecode Example: Step by Step

`int x = 5; System.out.println(x);` compiled (`javap -c` , JDK 21). The JVM is a **stack machine**: instructions push values onto the operand stack and pop them off.

```
1 0: iconst_5
2 1: istore_1
3 2: getstatic     #7    // System.out
▶ 5: iload_1
5 6: invokevirtual #13   // println(int)
6 9: return
```



`iload_1` : `x` 's value (5) is copied onto the stack. (Source: `System.out.println(x);`)

Step 4 / 6

VARIABLES

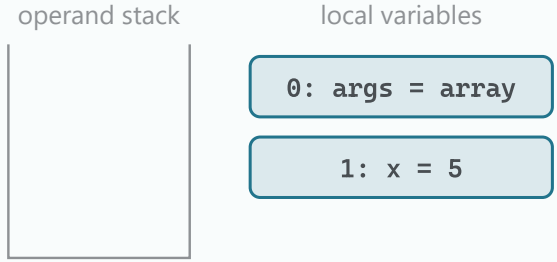
`x = 5` `stack = [System.out, 5]`

OUTPUT

Bytecode Example: Step by Step

`int x = 5; System.out.println(x);` compiled (`javap -c` , JDK 21). The JVM is a **stack machine**: instructions push values onto the operand stack and pop them off.

```
1 0: iconst_5
2 1: istore_1
3 2: getstatic     #7    // System.out
4 5: iload_1
▶ 6: invokevirtual #13   // println(int)
6 9: return
```



`invokevirtual #13` : `println(int)` is called and pops two values; **5** is printed.

Step 5 / 6

VARIABLES

`x = 5` `stack = []`

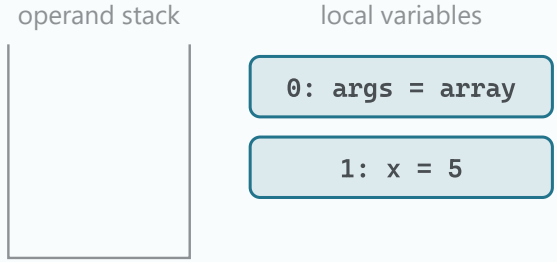
OUTPUT

```
5
```

Bytecode Example: Step by Step

`int x = 5; System.out.println(x);` compiled (`javap -c` , JDK 21). The JVM is a **stack machine**: instructions push values onto the operand stack and pop them off.

```
1 0: iconst_5
2 1: istore_1
3 2: getstatic     #7    // System.out
4 5: iload_1
5 6: invokevirtual #13   // println(int)
▶ 9: return
```



`return` : `main` ends, and the program terminates.

Step 6 / 6

VARIABLES

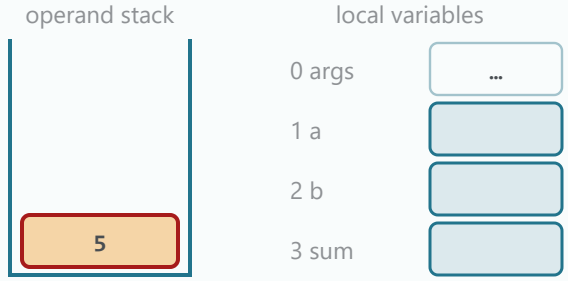
`x = 5` `stack = []`

OUTPUT

```
5
```

Bytecode: The JVM Stack Step by Step

```
▶ iconst_5          // push 5
2 istore_1          // a = 5
3 bipush 7          // push 7
4 istore_2          // b = 7
5 iload_1           // load a
6 iload_2           // load b
7 iadd              // add
8 istore_3          // sum = 12
9 getstatic         // System.out
10 iload_3          // load sum
11 invokevirtual    // println(int)
12 return
```



The JVM uses a **stack**, not registers. `iconst_5` pushes the constant 5 onto the operand stack.

Step 1 / 12

VARIABLES

`a = –` `b = –` `sum = –`

OUTPUT

Bytecode: The JVM Stack Step by Step

```
1  iconst_5        // push 5
▶ istore_1        // a = 5
3  bipush 7        // push 7
4  istore_2        // b = 7
5  iload_1         // load a
6  iload_2         // load b
7  iadd            // add
8  istore_3        // sum = 12
9  getstatic       // System.out
10 iload_3         // load sum
11 invokevirtual   // println(int)
12 return
```



`istore_1` pops the top of the stack into local 1 (a).

Step 2 / 12

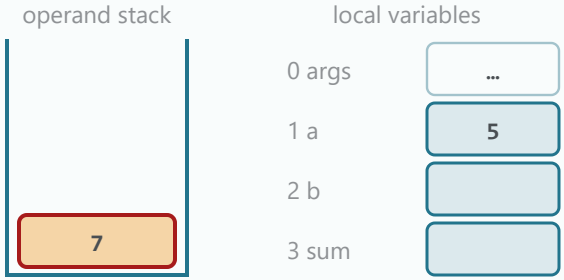
VARIABLES

`a = 5` `b = –` `sum = –`

OUTPUT

Bytecode: The JVM Stack Step by Step

```
1  iconst_5        // push 5
2  istore_1        // a = 5
▶  bipush 7        // push 7
4  istore_2        // b = 7
5  iload_1         // load a
6  iload_2         // load b
7  iadd            // add
8  istore_3        // sum = 12
9  getstatic       // System.out
10 iload_3         // load sum
11 invokevirtual   // println(int)
12 return
```



`bipush 7` : 7 fits in a byte; it is pushed onto the stack.

Step 3 / 12

VARIABLES

`a = 5` `b = –` `sum = –`

OUTPUT

Bytecode: The JVM Stack Step by Step

```
1  iconst_5        // push 5
2  istore_1        // a = 5
3  bipush 7        // push 7
▶  istore_2        // b = 7
5  iload_1         // load a
6  iload_2         // load b
7  iadd            // add
8  istore_3        // sum = 12
9  getstatic       // System.out
10 iload_3         // load sum
11 invokevirtual   // println(int)
12 return
```



7 is stored in local 2 (b).

Step 4 / 12

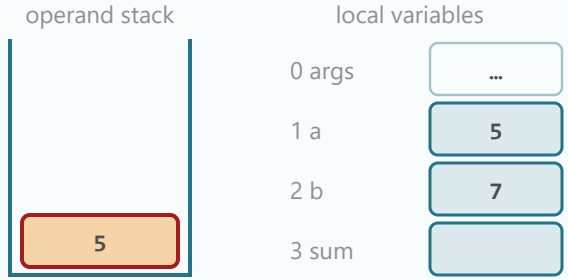
VARIABLES

`a = 5` `b = 7` `sum = –`

OUTPUT

Bytecode: The JVM Stack Step by Step

```
1  iconst_5        // push 5
2  istore_1        // a = 5
3  bipush 7        // push 7
4  istore_2        // b = 7
▶ iload_1          // load a
6  iload_2         // load b
7  iadd            // add
8  istore_3        // sum = 12
9  getstatic       // System.out
10 iload_3         // load sum
11 invokevirtual   // println(int)
12 return
```



`iload_1` : a's value (5) is copied onto the stack; a is unchanged.

Step 5 / 12

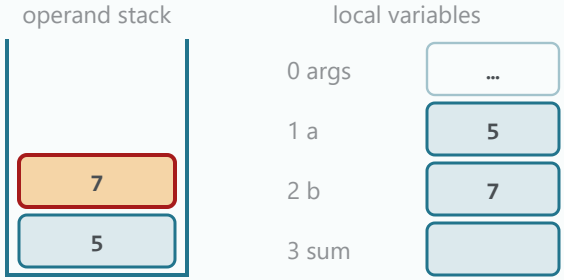
VARIABLES

`a = 5` `b = 7` `sum = –`

OUTPUT

Bytecode: The JVM Stack Step by Step

```
1  iconst_5        // push 5
2  istore_1        // a = 5
3  bipush 7        // push 7
4  istore_2        // b = 7
5  iload_1         // load a
▶  iload_2         // load b
7  iadd            // add
8  istore_3        // sum = 12
9  getstatic       // System.out
10 iload_3         // load sum
11 invokevirtual   // println(int)
12 return
```



b's value (7) goes on top. Two values are on the stack.

Step 6 / 12

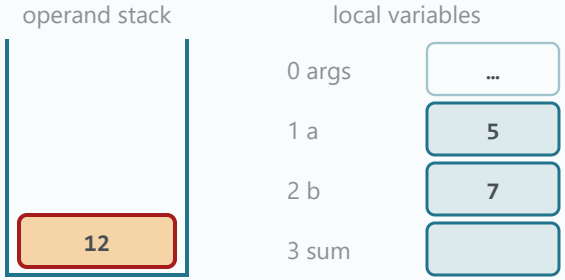
VARIABLES

`a = 5` `b = 7` `sum = –`

OUTPUT

Bytecode: The JVM Stack Step by Step

```
1  iconst_5        // push 5
2  istore_1        // a = 5
3  bipush 7        // push 7
4  istore_2        // b = 7
5  iload_1         // load a
6  iload_2         // load b
▶  iadd            // add
8  istore_3        // sum = 12
9  getstatic       // System.out
10 iload_3         // load sum
11 invokevirtual   // println(int)
12 return
```



`iadd` pops the top two values, adds them and pushes the result (12).

Step 7 / 12

VARIABLES

`a = 5` `b = 7` `sum = –`

OUTPUT

Bytecode: The JVM Stack Step by Step

```
1  iconst_5        // push 5
2  istore_1        // a = 5
3  bipush 7        // push 7
4  istore_2        // b = 7
5  iload_1         // load a
6  iload_2         // load b
7  iadd            // add
▶  istore_3        // sum = 12
9  getstatic       // System.out
10 iload_3         // load sum
11 invokevirtual   // println(int)
12 return
```



12 is stored in local 3 (sum): `int sum = a + b;` is done.

Step 8 / 12

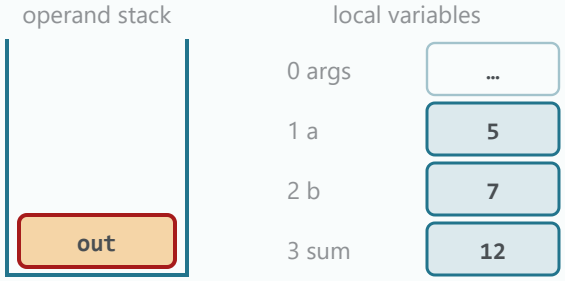
VARIABLES

`a = 5` `b = 7` `sum = 12`

OUTPUT

Bytecode: The JVM Stack Step by Step

```
1  iconst_5        // push 5
2  istore_1        // a = 5
3  bipush 7        // push 7
4  istore_2        // b = 7
5  iload_1         // load a
6  iload_2         // load b
7  iadd            // add
8  istore_3        // sum = 12
▶  getstatic       // System.out
10 iload_3         // load sum
11 invokevirtual   // println(int)
12 return
```



The `System.out` reference is pushed onto the stack.

Step 9 / 12

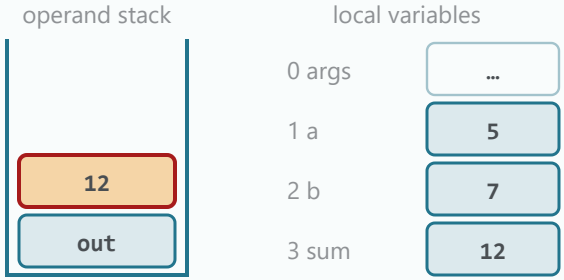
VARIABLES

`a = 5` `b = 7` `sum = 12`

OUTPUT

Bytecode: The JVM Stack Step by Step

```
 1  iconst_5        // push 5
 2  istore_1        // a = 5
 3  bipush 7        // push 7
 4  istore_2        // b = 7
 5  iload_1         // load a
 6  iload_2         // load b
 7  iadd            // add
 8  istore_3        // sum = 12
 9  getstatic       // System.out
▶10 iload_3         // load sum
11  invokevirtual   // println(int)
12  return
```



sum (12) is loaded onto the stack: the argument to println.

Step 10 / 12

VARIABLES

`a = 5` `b = 7` `sum = 12`

OUTPUT

Bytecode: The JVM Stack Step by Step

```
 1  iconst_5        // push 5
 2  istore_1        // a = 5
 3  bipush 7        // push 7
 4  istore_2        // b = 7
 5  iload_1         // load a
 6  iload_2         // load b
 7  iadd            // add
 8  istore_3        // sum = 12
 9  getstatic       // System.out
10  iload_3         // load sum
▶   invokevirtual   // println(int)
12  return
```



`println(int)` is called; the object and argument are popped, and 12 is printed.

Step 11 / 12

VARIABLES

`a = 5` `b = 7` `sum = 12`

OUTPUT

```
12
```

Bytecode: The JVM Stack Step by Step

```
1  iconst_5        // push 5
2  istore_1        // a = 5
3  bipush 7        // push 7
4  istore_2        // b = 7
5  iload_1         // load a
6  iload_2         // load b
7  iadd            // add
8  istore_3        // sum = 12
9  getstatic       // System.out
10 iload_3         // load sum
11 invokevirtual   // println(int)
▶  return
```



`main` ends. The same bytecode runs this way on the JVM of every operating system.

Step 12 / 12

VARIABLES

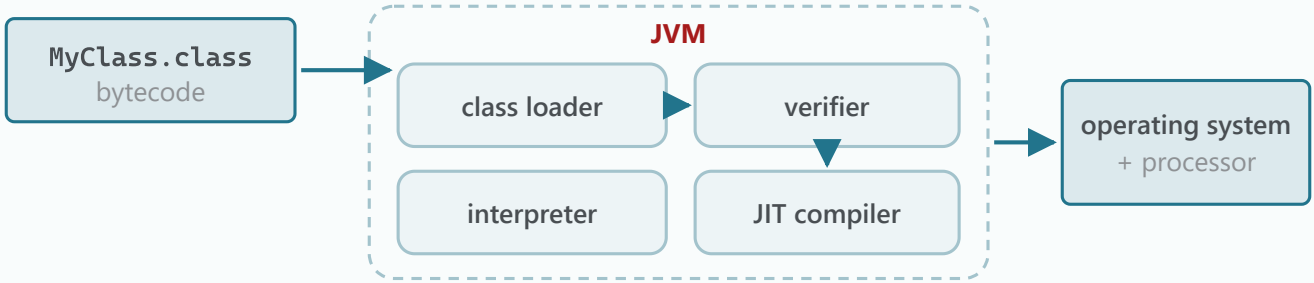
`a = 5` `b = 7` `sum = 12`

OUTPUT

```
12
```

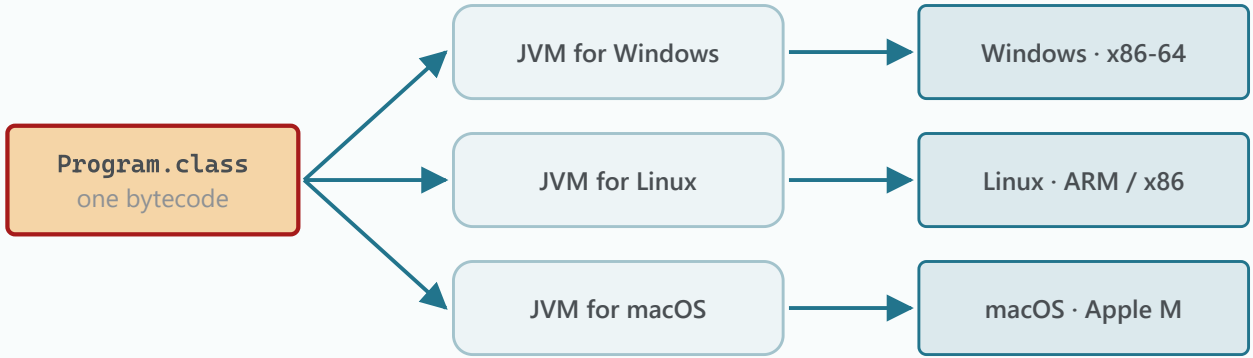
The Java Virtual Machine (JVM)

- The JVM (Java Virtual Machine) is the core component that runs Java applications: it **interprets** bytecode and runs it on the host processor.
- Frequently executed code is compiled to machine code at run time by the **JIT** (Just-In-Time) compiler, so programs speed up as they run.
- **The Java launcher** is the `java` command: it starts the JVM, loads the class and calls its `main` method.



Platform Independence

- One of Java's key features: **“Write once, run anywhere”**.
- A program is compiled once; the same bytecode runs on different operating systems.
- This works because the JVM has been written separately for many operating systems and processors.
- In languages like C, each platform needs its own build.

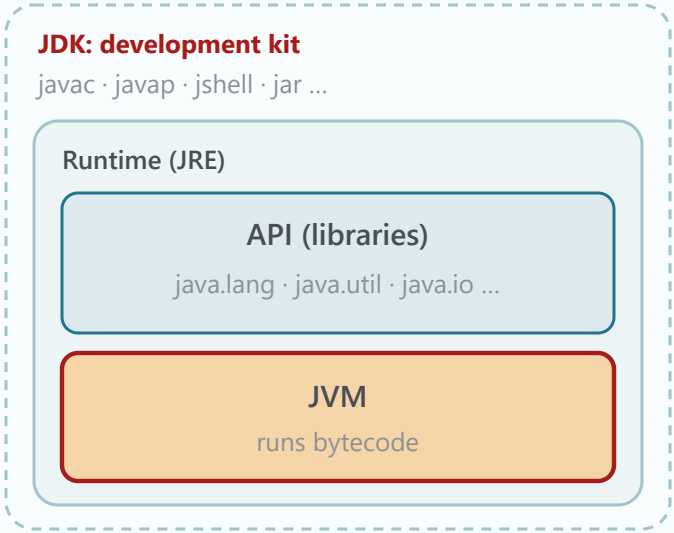


The Java Platform

Platform: the hardware or software environment a program runs in. The Java platform has two main parts:

- **JVM**: runs applications and makes them portable across hardware.
- **API**: ready-made class libraries for GUIs, databases, networking, files and much more; they speed up development.

To develop programs, you install the **JDK**. Since Java 11 the JRE is no longer shipped separately; the JDK includes everything.



Basic Programming Elements

Variables, types and expressions

Programs need variables to store and process data. Every variable has a **type**; values are computed by **expressions**: `num > 0`, `sum + 1`.

Control flow

Decides in what order and how the program runs:

- **Branching**: runs different parts depending on a condition, e.g. `if-else`.
- **Loops**: repeat an operation, e.g. `for`, `while`.

Sample code: variable + condition + output

```
int num = 5;
if (num > 0) {
    System.out.println("Positive.");
} else {
    System.out.println("Zero or negative.");
}
```

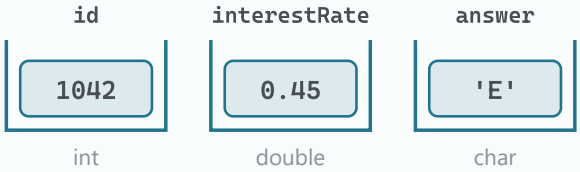
OUTPUT

```
Positive.
```

`num` is 5, so the condition is true; the `else` block is skipped.

Variables: The Building Blocks of Data Storage

- **A variable** is a memory location that holds a value and has a **name** and a **type**. Think of it as a **box** with a label on it.
- Every variable must be **declared** before use: type first, then name; a declaration ends with `;`.
- Several variables of the same type can be declared on one line, separated by commas.
- Choose descriptive, meaningful names: not `x` but `checkCount`.




```
int id, checkCount, stockCount;
double amount, interestRate;
char answer;
```

Syntax: `type name;` · `type name1, name2, name3;` · `type name = initialValue;`

Declaration, Assignment, Update

```
1 int age;
2 age = 25;
3 double height = 1.78;
4 age = age + 1;
5 char letter = 'K';
6 System.out.println(age + " " + height + " " +
  letter);
```



`int age;` reserves a box named `age` in memory; it is still **empty** (no value assigned).

Step 1 / 6

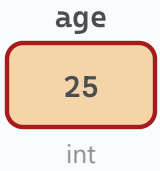
VARIABLES

`age = ?`

OUTPUT

Declaration, Assignment, Update

```
1 int age;
▶ age = 25;
3 double height = 1.78;
4 age = age + 1;
5 char letter = 'K';
6 System.out.println(age + " " + height + " " +
  letter);
```



Assignment: 25 goes into the box. Now `age` can be read.

Step 2 / 6

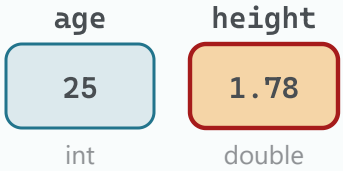
VARIABLES

`age = 25`

OUTPUT

Declaration, Assignment, Update

```
1 int age;
2 age = 25;
▶ double height = 1.78;
4 age = age + 1;
5 char letter = 'K';
6 System.out.println(age + " " + height + " " +
  letter);
```



Declaration and initialization on one line:

`double height = 1.78;`

Step 3 / 6

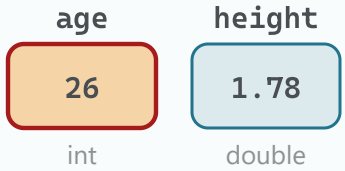
VARIABLES

`age = 25` `height = 1.78`

OUTPUT

Declaration, Assignment, Update

```
1 int age;
2 age = 25;
3 double height = 1.78;
▶ age = age + 1;
5 char letter = 'K';
6 System.out.println(age + " " + height + " " +
  letter);
```



The right side is evaluated first: 25 + 1 = 26; the result is stored in `age`, and the old value is lost.

Step 4 / 6

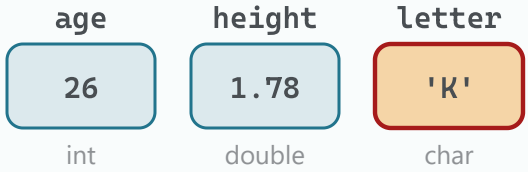
VARIABLES

`age = 26` `height = 1.78`

OUTPUT

Declaration, Assignment, Update

```
1 int age;
2 age = 25;
3 double height = 1.78;
4 age = age + 1;
▶ char letter = 'K';
6 System.out.println(age + " " + height + " " +
  letter);
```



`char` values are written in single quotes.

Step 5 / 6

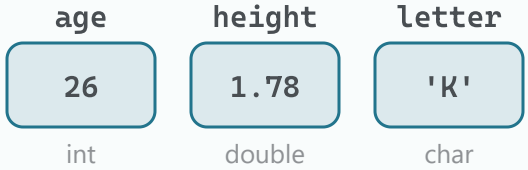
VARIABLES

`age = 26` `height = 1.78` `letter = 'K'`

OUTPUT

Declaration, Assignment, Update

```
1 int age;
2 age = 25;
3 double height = 1.78;
4 age = age + 1;
5 char letter = 'K';
▶ System.out.println(age + " " + height + " " +
  letter);
```



The values are joined to text with `+` and printed.

Step 6 / 6

VARIABLES

`age = 26` `height = 1.78` `letter = 'K'`

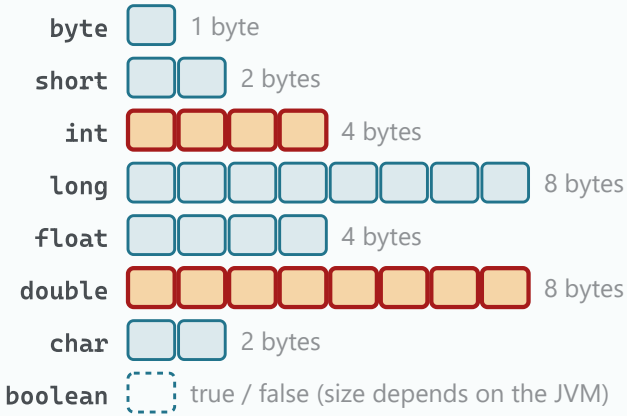
OUTPUT

```
26 1.78 K
```

Primitive Data Types

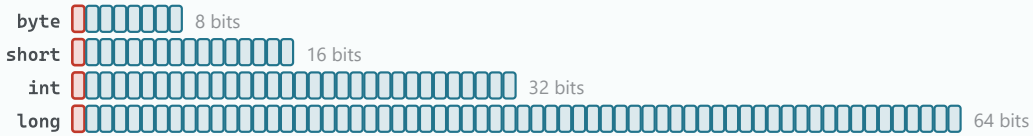
- Java has **eight** primitive data types; each has its own **memory size** and **range of values**.
- **Integer:** `byte`, `short`, `int`, `long`
- **Floating-point:** `float`, `double`
- **Character:** `char`
- **Boolean:** `boolean`

In everyday code, `int` is the default for integers and `double` for decimals.



Integer Types

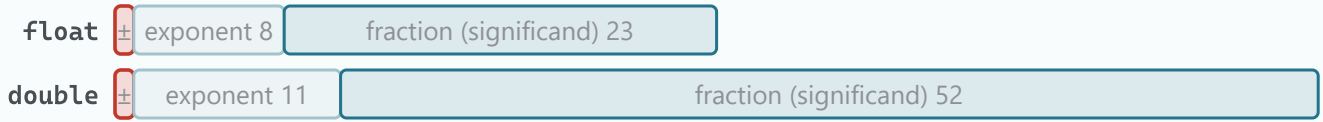
Type	Memory	Minimum	Maximum	Example
`byte`	1 byte (8 bits)	−128	127	`byte b = 100;`
`short`	2 bytes (16 bits)	−32,768	32,767	`short s = 32000;`
`int`	4 bytes (32 bits)	−2,147,483,648	2,147,483,647	`int n = 1_000_000;`
`long`	8 bytes (64 bits)	−9,223,372,036,854,775,808	9,223,372,036,854,775,807	`long l = 3000000000L;`



- All are **signed**: an n-bit type ranges from -2^{n-1} … $2^{n-1} - 1$. The limits are available as constants such as `Byte.MIN_VALUE`, `Integer.MAX_VALUE`.
- An integer literal such as `42` has type `int`; a `long` value beyond the `int` range needs the suffix `L`.

Floating-Point Types

Type	Memory	Smallest positive	Largest (±)	Significant digits
`float`	4 bytes (32 bits)	≈ 1.4×10^{-45}	≈ 3.4028235×10^{38}	≈ 7
`double`	8 bytes (64 bits)	≈ 4.9×10^{-324}	≈ $1.7976931348623157 \times 10^{308}$	≈ 15–16



- A decimal literal such as `3.14` is a `double`; for a `float`, add `f`: `3.14f`. A `float` holds ≈ 7 digits, so `1.1f * 3` → `3.3000002`.
- The "smallest" column is the smallest nonzero **positive** value (`Float.MIN_VALUE`, `Double.MIN_VALUE`); the negative limits are the negatives of the largest values.
- `double` is double precision: compared with `float`, it has both a wider range and more precision.

char and boolean

`char`: a single character

2 bytes (16 bits), **unsigned**. Range 0 … 65,535: each value is a **Unicode** character code. Written in single quotes: `'A'`, `'ş'`, `'7'`.

`'A'` = 65, `'a'` = 97, `'0'` = 48. Look up other codes in an ASCII table.

`boolean`: a logical value

It has only two values: `true` and `false`. It carries one bit of information; the JVM decides how many bytes it takes in memory.

Used in conditions and comparison results: `boolean passed = score >= 50;`

```
char letter = 'A';
boolean valid = true;
System.out.println(letter);
System.out.println(valid);
System.out.println(!valid);
```

OUTPUT

```
A
true
false
```

Seeing Type Limits in a Program

```
System.out.println("byte  : " + Byte.MIN_VALUE + " .. " + Byte.MAX_VALUE);
System.out.println("short : " + Short.MIN_VALUE + " .. " + Short.MAX_VALUE);
System.out.println("int   : " + Integer.MIN_VALUE + " .. " + Integer.MAX_VALUE);
System.out.println("long  : " + Long.MIN_VALUE + " .. " + Long.MAX_VALUE);
System.out.println("float : " + Float.MIN_VALUE + " .. " + Float.MAX_VALUE);
System.out.println("double: " + Double.MIN_VALUE + " .. " + Double.MAX_VALUE);
System.out.println("char  : " + (int) Character.MIN_VALUE + " .. "
                              + (int) Character.MAX_VALUE);
```

OUTPUT

```
byte  : -128 .. 127
short : -32768 .. 32767
int   : -2147483648 .. 2147483647
long  : -9223372036854775808 .. 9223372036854775807
float : 1.4E-45 .. 3.4028235E38
double: 4.9E-324 .. 1.7976931348623157E308
char  : 0 .. 65535
```

`4.9E-324` is scientific notation: 4.9×10^{-324}. `Integer`, `Double` and similar classes are the **wrapper** classes of the primitive types.

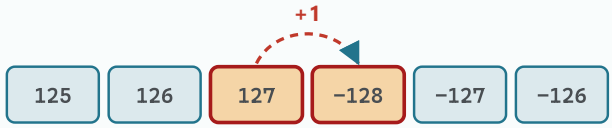
Overflow

```java
int max = Integer.MAX_VALUE;
System.out.println(max);
System.out.println(max + 1);    // overflow!
byte n1 = 80;
byte n2 = 50;
byte sum = (byte) (n1 + n2);
System.out.println(sum);         // not 130
long fixed = (long) max + 1;
System.out.println(fixed);
```

OUTPUT

```
2147483647
-2147483648
-126
2147483648
```

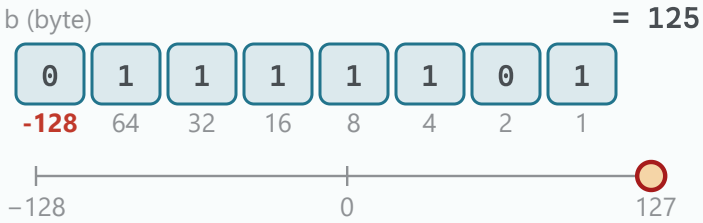
- A result beyond a type's limit **raises no error**: the value "wraps around" to the other end of the range (like a clock going from 12 to 1).
- 80 + 50 = 130, `byte` limit is 127 → 130 − 256 = **−126**.
- Fix: use a wider type (`long`) and convert **before** the operation.



byte: adding 1 to the maximum "wraps" to the minimum

Overflow: A byte Counter Step by Step

```
byte b = 125;
for (int i = 1; i <= 4; i++) {
    b++;
    System.out.println(b);
}
```



`b` is a `byte`: 8 bits. 125 in binary is `01111101`; the leftmost bit is the **sign bit** and has weight −128.

Step 1 / 8

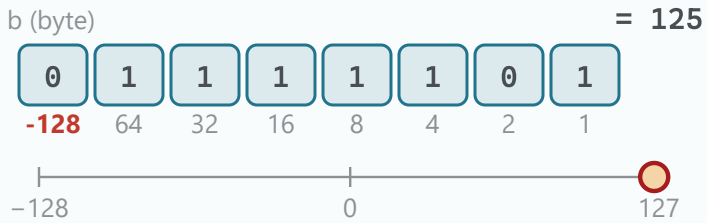
VARIABLES

b = **125**

OUTPUT

Overflow: A byte Counter Step by Step

```
1 byte b = 125;
▶ for (int i = 1; i <= 4; i++) {
3     b++;
4     System.out.println(b);
5 }
```



`i = 1` : `1 <= 4` is true; enter the loop body.

Step 2 / 8

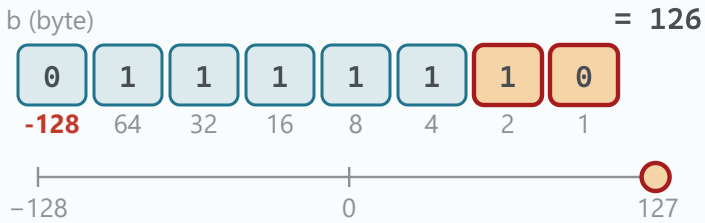
VARIABLES

`b = 125` `i = 1`

OUTPUT

Overflow: A byte Counter Step by Step

```
1 byte b = 125;
2 for (int i = 1; i <= 4; i++) {
      b++;
      System.out.println(b);
5 }
```



`b++` → 126 (`01111110`); the value is printed.

Step 3 / 8

VARIABLES

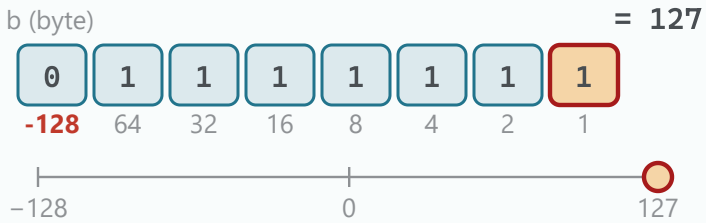
`b = 126` `i = 1`

OUTPUT

```
126
```

Overflow: A byte Counter Step by Step

```
1 byte b = 125;
2 for (int i = 1; i <= 4; i++) {
      b++;
      System.out.println(b);
5 }
```



`b++` → 127 (`01111111`): `byte` maximum. It is printed.

Step 4 / 8

VARIABLES

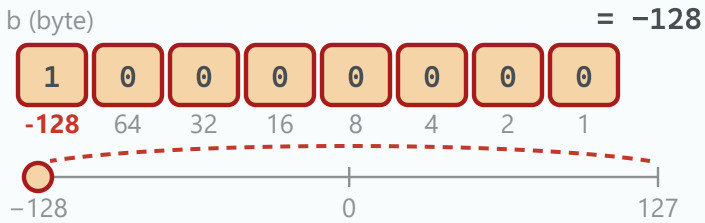
`b = 127` `i = 2`

OUTPUT

```
126
127
```

Overflow: A byte Counter Step by Step

```
1 byte b = 125;
2 for (int i = 1; i <= 4; i++) {
▶     b++;
4     System.out.println(b);
5 }
```



127 + 1: binary addition gives `10000000`. The sign bit is now 1 → the value is **−128**. No error: the value **wrapped around**.

Step 5 / 8

VARIABLES

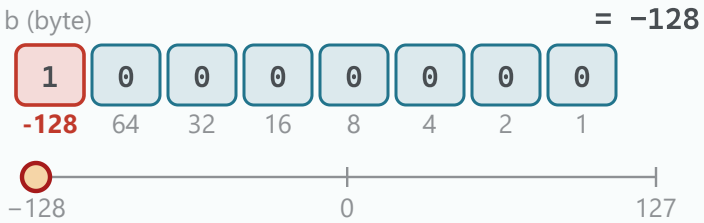
`b = −128` `i = 3`

OUTPUT

```
126
127
```

Overflow: A byte Counter Step by Step

```
1 byte b = 125;
2 for (int i = 1; i <= 4; i++) {
3     b++;
▶     System.out.println(b);
5 }
```



−128 is printed: the program carries on with a wrong result, without any warning.

Step 6 / 8

VARIABLES

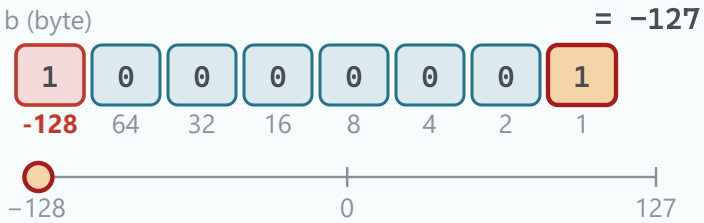
`b = −128` `i = 3`

OUTPUT

```
126
127
-128
```

Overflow: A byte Counter Step by Step

```
1 byte b = 125;
2 for (int i = 1; i <= 4; i++) {
      b++;
      System.out.println(b);
5 }
```



`b++` → −127 (`10000001`): counting now continues upward from −128.

Step 7 / 8

VARIABLES

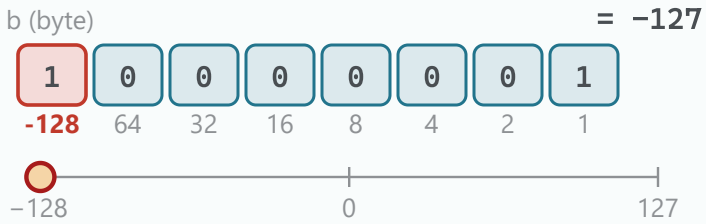
`b = -127` `i = 4`

OUTPUT

```
126
127
-128
-127
```

Overflow: A byte Counter Step by Step

```
1 byte b = 125;
▶ for (int i = 1; i <= 4; i++) {
3     b++;
4     System.out.println(b);
5 }
```



`i = 5` : `5 <= 4` is false; the loop ends. Output: 126, 127, −128, −127.

Step 8 / 8

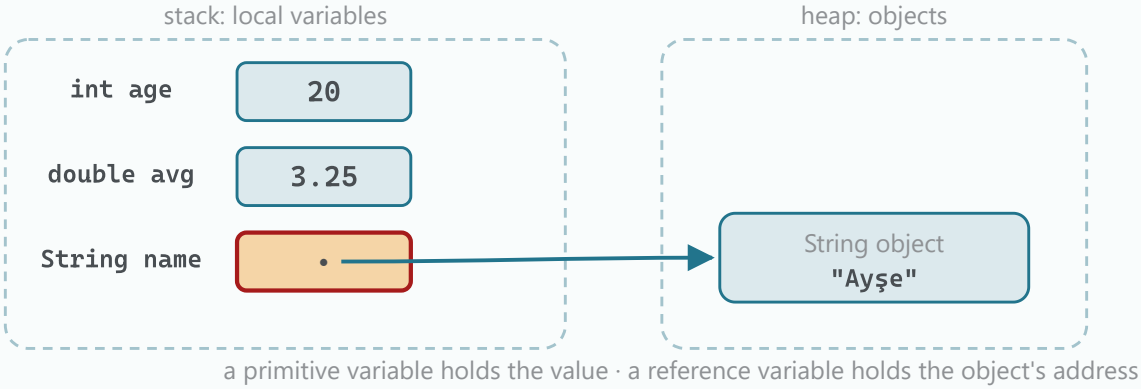
VARIABLES

`b = -127` `i = 5`

OUTPUT

```
126
127
-128
-127
```

Primitive Types and Reference Types



- **Primitive types** (8 of them): store the value itself; their names are lowercase.
- **Reference types**: refer to an **object**; the variable holds not the object itself but its location (a reference).
- Examples: `String` (character strings), arrays, classes, interfaces, `enum` s. Objects are usually created with `new`.
- A reference that points to no object has the value `null`. The empty string `""` is an object (`length()` = 0); `null` is different: calling a method on it throws `NullPointerException`.

Identifiers

- Names given to variables, classes and methods are called **identifiers**. They follow specific rules:
- May contain only **letters**, **digits** (0–9), the underscore `_` and the dollar sign `$`. No spaces, or characters such as `-`, `@`, `.`.
- The first character must be a letter, `_` or `$`; **never a digit**.
- Keywords and `true`, `false`, `null` cannot be identifiers.
- There is no length limit.
- Java is case-sensitive: `personName` and `personname` are **different** variables; a misspelled name gives a `cannot find symbol` error.

Valid examples

`userName`, `totalScore`, `student_name`, `data1`, `_count`

Non-English letters

Java accepts any Unicode letter (`öğrenci` compiles), but stick to English letters in names to avoid problems on other systems.

Which Identifiers Are Valid?

Name		Why?
`studentNo`	✓	starts with a letter
`num_2`	✓	letters, digits and `_`
`$price`	✓	`$` allowed (but discouraged)
`2count`	✗	starts with a digit
`pay rate`	✗	contains a space
`first-name`	✗	`-` is the minus operator
`@name`	✗	`@` not allowed
`class`	✗	keyword
`Class`	✓	capital C: not a keyword

A name starting with a digit confuses the compiler; the message points at the statement, not the name:

```
int 2count = 20;
```

```
Ident.java:3: error: not a statement
        int 2count = 20;
        ^
Ident.java:3: error: ';' expected
        int 2count = 20;
           ^
2 errors
```

Keywords (Reserved Words)

- Words with a specific role in the language; they are **reserved**. All are lowercase.
- They **cannot be used** as names of variables, classes or methods. Java 21 has 51 keywords:

`boolean` `byte` `char` `short` `int` `long` `float` `double` `void` `if` `else` `switch`
`case` `default` `for` `while` `do` `break` `continue` `return` `class` `interface` `enum`
`extends` `implements` `new` `this` `super` `instanceof` `package` `import` `public`
`protected` `private` `static` `final` `abstract` `native` `synchronized` `volatile`
`transient` `strictfp` `try` `catch` `finally` `throw` `throws` `assert` `const` `goto` `_`

types · control flow · classes and objects · modifiers · exceptions · reserved, unused

`true`, `false`, `null` are not keywords but **literals**; still, they cannot be names either. `var` (Java 10+), `record` (16+), `yield` (14+) and other **contextual** keywords have special meaning only in certain places.

Keywords: Data Types

Keyword	Purpose
`boolean`	Only two values: `true` and `false`. Used in conditions and logical operations.
`byte`	8-bit (1-byte) signed integer; for small integers.
`short`	16-bit integer; uses less memory than `int` for smaller numbers.
`int`	32-bit integer; the default integer type in calculations.
`long`	64-bit integer; for large integers that don't fit in an `int`.
`float`	32-bit single-precision floating point; less precise than `double`.
`double`	64-bit double-precision floating point; for large and precise decimal values.
`char`	A single 16-bit Unicode character.
`void`	Indicates that a method **returns no value**: `public static void main`.

Keywords: Control Flow

Keyword	Purpose
`if / else`	Tests a condition; if it is true, the `if` block runs, otherwise the `else` block runs.
`switch`	Performs different actions for different values of a variable.
`case`	`switch` body: marks one case (value).
`default`	`switch` body: what to do when no case matches.
`for`	Repeats an operation under a condition, using a counter.
`while`	Repeats an operation while a condition holds.
`do`	`do-while` loop: the body runs at least once; the condition is checked at the end.
`break`	Exits a loop or a `switch` block immediately.
`continue`	Skips the rest of the current iteration and moves on to the next.
`return`	Ends a method, returning a result if needed.

Keywords: Classes and Objects

Keyword	Purpose
`class`	Defines a class. Classes are templates for objects and the basic building blocks of Java programs.
`interface`	Defines an interface: a contract listing the methods a class must provide.
`enum`	A type made of constant values: `enum Color { RED, GREEN }`.
`extends`	Inheritance: a class takes on the fields and methods of another class.
`implements`	A class implements an interface by providing its methods.
`new`	Creates an object: allocates memory and sets initial values.
`this`	A reference to the object itself; tells a field apart from a local variable with the same name.
`super`	Accesses the superclass's methods, fields and constructor.
`instanceof`	Checks whether an object is of a given class or interface type.
`package / import`	`package` sets the class's package (namespace); `import` makes classes from other packages available.

Keywords: Modifiers

Keyword	Purpose
`public`	Accessible from everywhere, by all classes.
`protected`	Accessible within the same package and from subclasses.
`private`	Accessible only within the same class.
`static`	The member belongs to the **class**, not to an object; it is shared by all objects.
`final`	Variable: value cannot change · method: cannot be overridden · class: cannot be extended.
`abstract`	Abstract class (cannot be instantiated) or a method without a body; subclasses must implement it.
`native`	A method whose body is written outside Java in platform-specific code (C/C++), linked via JNI.
`synchronized`	In multithreading, lets only one thread in at a time.
`volatile`	Makes every thread always read the variable's latest value.
`transient`	Excludes the field when the object is serialized (saved).
`strictfp`	Makes floating-point results identical on every platform. Since Java 17 this is the default, so the keyword has no effect.

Keywords: Exceptions and Others

Keyword	Purpose
`try`	Wraps code that may fail, so the exceptions it throws can be caught.
`catch`	`try` block: catches and handles the exception thrown there.
`finally`	`try-catch`: a block that always runs at the end (e.g. to close a file).
`throw`	Throws an exception object; normal flow stops.
`throws`	In a method header, lists the exceptions the method may throw.
`assert`	Checks that a condition holds; used for debugging and verification (enabled with `java -ea`).
`const, goto`	Reserved but unused; cannot be used as names.
`_`	Since Java 9, a lone underscore cannot be a name (Java 22+ gave it the meaning "unused variable").

```
int new = 5;
```

```
Keyword.java:3: error: not a statement
        int new = 5;
        ^
... (4 errors in total)
```

Naming Conventions

What?	Rule	Examples
Classes and interfaces	PascalCase, starts with uppercase	`String`, `Student`, `HelloWorld`
Variables, fields, methods	camelCase, starts with lowercase	`firstName`, `classAverage`, `compute()`
Primitive data types	lowercase (they are keywords)	`int`, `float`, `boolean`
Named constants	UPPER_CASE, words joined with `_`	`PI`, `DAYS_PER_WEEK`, `MAX_COUNT`
Package	all lowercase	`syntax`, `tr.edu.giresun`

camelCase: first word lowercase, later words capitalized (like a camel's humps). **PascalCase**: every word starts with a capital. These are not compiler rules but **conventions**: break them and the code still compiles, but it is harder to read.

```
class StudentRecord { }    // class
String fullName;       // variable
boolean isMarried;           // yes/no
final int MAX_COUNT = 100; // constant
```

Assignment Statements

- To give a variable a value, use the `=` **assignment operator**: `variable = expression;`
- First the **right-hand** expression is evaluated, and the result is stored in the **left-hand** variable.
- The expression can be a literal, another variable or an operation.
- It is not equality as in math: `x = x + 1` means "the new value of x is its old value plus one".
- Only a variable may appear on the left: `5 = num;` does not compile (`unexpected type`).

```
int num, result;
int num1 = 4, num2 = 6;
String name;

num = 10;              // literal
result = num1 + num2; // sum: 10
name = "Ahmet";         // text
num = num * 2;        // 10 → 20
```



Types must match in an assignment: no number into a `String` variable, no text into an `int` variable.

Compound Assignment Operators: Trace

`+=` `-=` `*=` `/=` `%=` : `x op= y` ↔ `x = x op y` . Shorter to write, same effect.

```
1 int x = 5;
2 int y = 10;
3 x = x + y;
4 x -= y;
5 x *= 2;
6 x /= 4;
7 x %= 3;
8 System.out.println("x = " + x + ", y = " + y);
```



`x` starts at 5.

Step 1 / 8

VARIABLES

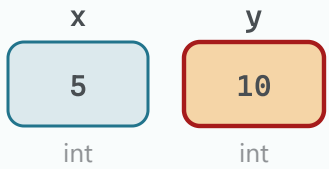
x = 5

OUTPUT

Compound Assignment Operators: Trace

`+=` `-=` `*=` `/=` `%=` : `x op= y` ↔ `x = x op y` . Shorter to write, same effect.

```
1 int x = 5;
▶ int y = 10;
3 x = x + y;
4 x -= y;
5 x *= 2;
6 x /= 4;
7 x %= 3;
8 System.out.println("x = " + x + ", y = " + y);
```



`y` starts at 10.

Step 2 / 8

VARIABLES

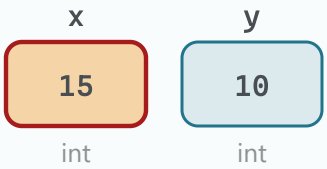
`x = 5` `y = 10`

OUTPUT

Compound Assignment Operators: Trace

`+=` `-=` `*=` `/=` `%=` : `x op= y` ↔ `x = x op y`. Shorter to write, same effect.

```
1 int x = 5;
2 int y = 10;
▶ x = x + y;
4 x -= y;
5 x *= 2;
6 x /= 4;
7 x %= 3;
8 System.out.println("x = " + x + ", y = " + y);
```



Right side: 5 + 10 = 15 → `x = 15`.

Step 3 / 8

VARIABLES

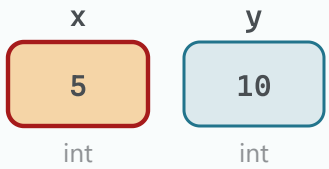
`x = 15` `y = 10`

OUTPUT

Compound Assignment Operators: Trace

`+=` `-=` `*=` `/=` `%=` : `x op= y` ↔ `x = x op y`. Shorter to write, same effect.

```
1 int x = 5;
2 int y = 10;
3 x = x + y;
▶ x -= y;
5 x *= 2;
6 x /= 4;
7 x %= 3;
8 System.out.println("x = " + x + ", y = " + y);
```



`x -= y` is short for `x = x - y`: 15 − 10 = 5.

Step 4 / 8

VARIABLES

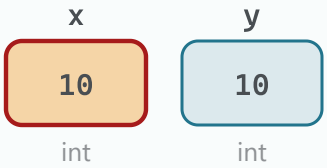
`x = 5` `y = 10`

OUTPUT

Compound Assignment Operators: Trace

`+=` `-=` `*=` `/=` `%=` : `x op= y` ↔ `x = x op y` . Shorter to write, same effect.

```
1 int x = 5;
2 int y = 10;
3 x = x + y;
4 x -= y;
▶ x *= 2;
6 x /= 4;
7 x %= 3;
8 System.out.println("x = " + x + ", y = " + y);
```



`x = x * 2` : 5 × 2 = 10.

Step 5 / 8

VARIABLES

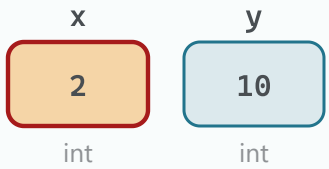
`x = 10` `y = 10`

OUTPUT

Compound Assignment Operators: Trace

`+=` `-=` `*=` `/=` `%=` : `x op= y` ↔ `x = x op y` . Shorter to write, same effect.

```
1 int x = 5;
2 int y = 10;
3 x = x + y;
4 x -= y;
5 x *= 2;
▶ x /= 4;
7 x %= 3;
8 System.out.println("x = " + x + ", y = " + y);
```



`x = x / 4` : 10 / 4 = 2 (integer division, fraction dropped).

Step 6 / 8

VARIABLES

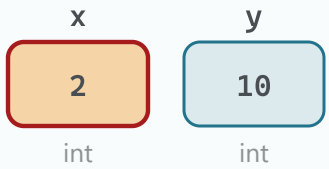
`x = 2` `y = 10`

OUTPUT

Compound Assignment Operators: Trace

`+=` `-=` `*=` `/=` `%=` : `x op= y` ↔ `x = x op y` . Shorter to write, same effect.

```
1 int x = 5;
2 int y = 10;
3 x = x + y;
4 x -= y;
5 x *= 2;
6 x /= 4;
▶ x %= 3;
8 System.out.println("x = " + x + ", y = " + y);
```



`x = x % 3` : the remainder of 2 divided by 3 is 2.

Step 7 / 8

VARIABLES

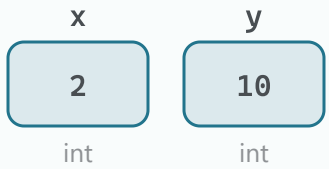
`x = 2` `y = 10`

OUTPUT

Compound Assignment Operators: Trace

`+=` `-=` `*=` `/=` `%=` : `x op= y` ↔ `x = x op y` . Shorter to write, same effect.

```
1 int x = 5;
2 int y = 10;
3 x = x + y;
4 x -= y;
5 x *= 2;
6 x /= 4;
7 x %= 3;
▶ System.out.println("x = " + x + ", y = " + y);
```



The final values are printed.

Step 8 / 8

VARIABLES

`x = 2` `y = 10`

OUTPUT

```
x = 2, y = 10
```

Initializing Variables

- A variable can be declared without a value, or initialized in its declaration: `int age = 25;`
- **Local variables** (inside methods) **get no default value**: reading one before assigning it is a compile error.
- Class **fields** get default values automatically: numbers 0, `boolean` `false`, `char` `'\u0000'`; reference types (e.g. `String`) `null`.
- Still, it is good practice not to rely on defaults and to assign values explicitly.

```
int count;
System.out.println(count);
```

```
Init.java:4: error: variable count
  might not have been initialized
```

Field type	Default
`int`, `long`, …	`0`
`double`, `float`	`0.0`
`char`	`'\u0000'` (code 0)
`boolean`	`false`
`String` and other references	`null`

Literals

Values written directly in code, such as `2`, `3.7`, `'y'`, `"Ali"`, are called **literals**.

- **Integer:** may start with `+` or `-`; **no commas**: `+42`, `-17`. Underscores aid readability (Java 7+): `1_000_000`. Hex `0x1F`, binary `0b1010`.
- **Floating-point:** written with a decimal point; `e` notation works too: `2.5e3` = 2.5 × 10^{3}, `7.68e-2` = 7.68 × 10^{-2}.
- **Character:** `'y'`; special characters: `'\n'` newline, `'\t'` tab, `'\''`, `'\\'`.
- **Text:** `"Hello"` · **boolean:** `true`, `false`

```
System.out.println(2.5e3);
System.out.println(7.68e-2);
System.out.println(1_000_000);
System.out.println(0x1F);
System.out.println(0b1010);
System.out.println(3000000000L);
System.out.println("Ali\tVeli\n\"Java\"");
```

OUTPUT

```
2500.0
0.0768
1000000
31
10
3000000000
Ali     Veli
"Java"
```

Floating-Point Precision

- Floating-point numbers are stored in a limited number of bits, in **binary**.
- Values like 1/3 or 0.1 have infinitely many digits in binary (0.1 = 0.000110011...$_2$, just as 1/3 = 0.333...), so they are stored **approximately**: `1.0/3.0` is not exactly 1/3.
- Small rounding errors add up: adding 0.1 ten times does not give exactly 1. Sometimes errors cancel out: the sum of three `1.0/3.0` terms prints `1.0`.
- So floating-point numbers are not compared with `==`; instead, check whether the difference is tiny. For money, use `BigDecimal`.

```
System.out.println(1.0 / 3.0);
System.out.println(1.0/3 + 1.0/3 + 1.0/3);
System.out.println(0.1 + 0.2);
System.out.println(0.1 + 0.2 == 0.3);
double t = 0.0;
for (int i = 0; i < 10; i++) t += 0.1;
System.out.println(t);
double diff = (0.1 + 0.2) - 0.3;
System.out.println(Math.abs(diff) < 1e-9);
```

OUTPUT

```
0.3333333333333333
1.0
0.30000000000000004
false
0.9999999999999999
true
```

Named Constants

- `final` variables get their value once; after that it **cannot change**. Pattern: `public static final Type CONSTANT_NAME = value;`
- Names are written in UPPERCASE, with words separated by `_`. Use meaningful names instead of "magic numbers": not `7` but `DAYS_PER_WEEK`; if the value changes, you update it in one place.

```java
public class Constants {
    public static final double PI = 3.14159;
    public static final int DAYS_PER_WEEK = 7;

    public static void main(String[] args) {
        double r = 2.0;
        double area = PI * r * r;
        System.out.println("Area: " + area);
        int days = 4 * DAYS_PER_WEEK;
        System.out.println(days + " days");
    }
}
```

OUTPUT

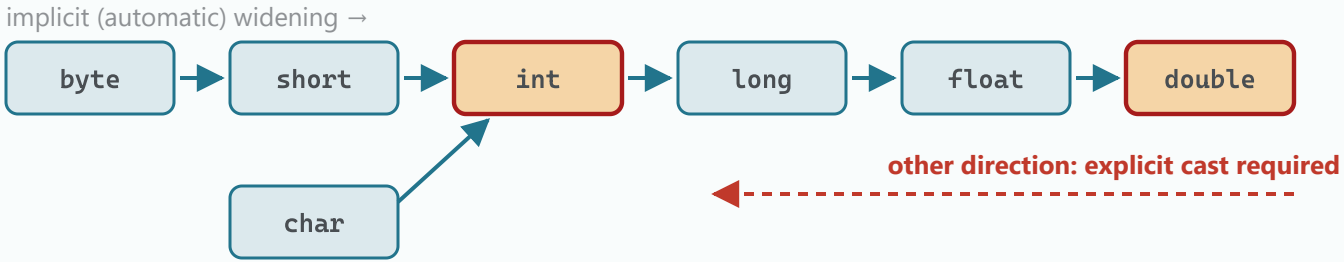
```
Area: 12.56636
28 days
```

Changing a constant, or assigning a `double` result to a `float`, does not compile:

```java
PI = 3.14;
float area = PI * r * r;
```

```
error: cannot assign a value to
  static final variable PI
error: incompatible types: possible
  lossy conversion from double to float
```

Assignment Compatibility



- A value can be assigned directly to a variable of any type **further right** in the chain (widening).
- `char` values can go into `int` (and the types to its right); not the reverse.
- `long` → `float` is allowed, but very large numbers may lose some digits to rounding.

```
char ch = 'A';
int n;
n = ch;   // OK: 65
ch = n;   // error
```

```
Compat.java:5: error: incompatible types:
  possible lossy conversion from int to char
        ch = n;
             ^
```

Data Type Conversion (Casting)

Implicit conversion

From a smaller to a larger type, Java converts **automatically**; the value is preserved (very large integers may be rounded when going to `float`).

Explicit conversion

From a larger to a smaller type (narrowing), you ask for the conversion by writing the target type **in parentheses**: `(int) d`. The programmer accepts any loss:

- decimal to integer: the fractional part is **dropped** (not rounded),
- an integer that doesn't fit overflows (`(byte) 130` → −126).

```java
double dbl = 5;     // implicit: 5.0
System.out.println(dbl);
double d = 5.0;
int num = (int) d;   // explicit: 5
System.out.println(num);
System.out.println((int) 3.99);
System.out.println((int) -3.99);
System.out.println(Math.round(3.99));
System.out.println((byte) 130);
```

OUTPUT

```
5.0
5
3
-3
4
-126
```

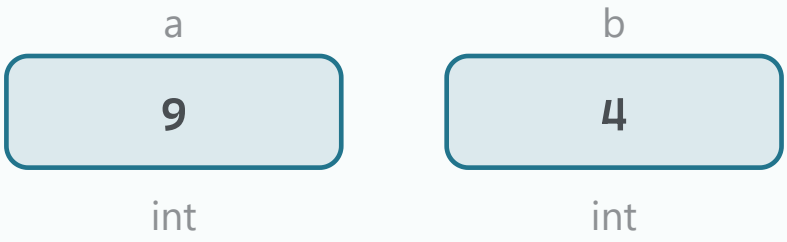
Integer Division and Where the Cast Goes

Expression	Result	Why?
`7 / 2`	`3`	two `int` → integer division, fraction dropped
`7 / 2.0`	`3.5`	`2.0` `double` → computed in `double`
`(double) 7 / 2`	`3.5`	7 → 7.0 first, then divide
`(double) (7 / 2)`	`3.0`	integer division first (3), then 3 → 3.0: the cast comes too late
`(int) 3.99`	`3`	fraction dropped
`Math.round(3.99)`	`4`	rounds to the nearest integer (returns a `long`)

An operation on two `int` values gives an `int`. If one operand is a `double`, the other is promoted to `double` too (numeric promotion). `byte` and `short` values are promoted to `int` before any arithmetic: `byte + byte` gives an `int`.

Type Conversion: Tracing the Code

```
1 int a = 9, b = 4;
2 double d1 = a / b;
3 double d2 = (double) a / b;
4 double d3 = (double) (a / b);
5 int k = (int) 9.75;
6 long r = Math.round(9.75);
7 byte y = (byte) 200;
8 System.out.println(d1 + " " + d2 + " " + d3);
9 System.out.println(k + " " + r + " " + y);
```



Two `int` variables: `a = 9`, `b = 4`.

Step 1 / 9

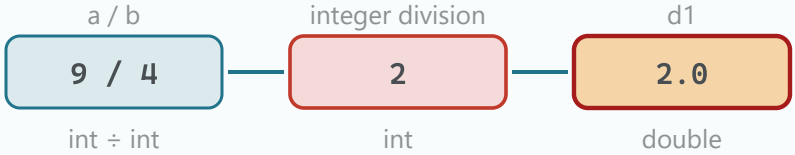
VARIABLES

`a = 9` `b = 4`

OUTPUT

Type Conversion: Tracing the Code

```
1 int a = 9, b = 4;
▶ double d1 = a / b;
3 double d2 = (double) a / b;
4 double d3 = (double) (a / b);
5 int k = (int) 9.75;
6 long r = Math.round(9.75);
7 byte y = (byte) 200;
8 System.out.println(d1 + " " + d2 +
  " " + d3);
9 System.out.println(k + " " + r + "
  " + y);
```



Two `int` values are divided: the result is **2** (fraction dropped). The conversion happens **afterwards**: 2 → 2.0. A `double` target does not save the division.

Step 2 / 9

VARIABLES

`a = 9` `b = 4` `d1 = 2.0`

OUTPUT

Type Conversion: Tracing the Code

```
1 int a = 9, b = 4;
2 double d1 = a / b;
▶ double d2 = (double) a / b;
4 double d3 = (double) (a / b);
5 int k = (int) 9.75;
6 long r = Math.round(9.75);
7 byte y = (byte) 200;
8 System.out.println(d1 + " " + d2 +
  " " + d3);
9 System.out.println(k + " " + r + "
  " + y);
```



First, `(double) a` = 9.0; since one operand is a `double`, `b` is promoted to 4.0 too: 9.0 / 4.0 = **2.25**.

Step 3 / 9

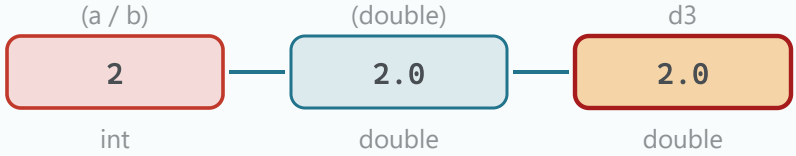
VARIABLES

`a = 9` `b = 4` `d1 = 2.0` `d2 = 2.25`

OUTPUT

Type Conversion: Tracing the Code

```
1 int a = 9, b = 4;
2 double d1 = a / b;
3 double d2 = (double) a / b;
▶ double d3 = (double) (a / b);
5 int k = (int) 9.75;
6 long r = Math.round(9.75);
7 byte y = (byte) 200;
8 System.out.println(d1 + " " + d2 +
  " " + d3);
9 System.out.println(k + " " + r + "
  " + y);
```



Parentheses first: `a / b` = 2; then 2 → 2.0. The cast comes too late.

Step 4 / 9

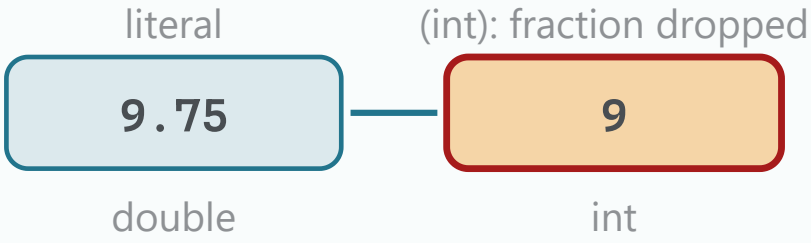
VARIABLES

`a = 9` `b = 4` `d1 = 2.0` `d2 = 2.25` `d3 = 2.0`

OUTPUT

Type Conversion: Tracing the Code

```
1 int a = 9, b = 4;
2 double d1 = a / b;
3 double d2 = (double) a / b;
4 double d3 = (double) (a / b);
▶ int k = (int) 9.75;
6 long r = Math.round(9.75);
7 byte y = (byte) 200;
8 System.out.println(d1 + " " + d2 + " " + d3);
9 System.out.println(k + " " + r + " " + y);
```



`(int) 9.75` : the fractional part is **dropped**, not rounded → 9.

Step 5 / 9

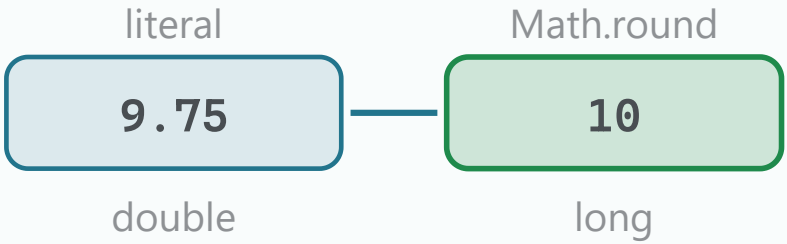
VARIABLES

`d1 = 2.0` `d2 = 2.25` `d3 = 2.0` `k = 9`

OUTPUT

Type Conversion: Tracing the Code

```
1 int a = 9, b = 4;
2 double d1 = a / b;
3 double d2 = (double) a / b;
4 double d3 = (double) (a / b);
5 int k = (int) 9.75;
▶ long r = Math.round(9.75);
7 byte y = (byte) 200;
8 System.out.println(d1 + " " + d2 + " " + d3);
9 System.out.println(k + " " + r + " " + y);
```



`Math.round(9.75)` gives the nearest integer: **10** (`long` type).

Step 6 / 9

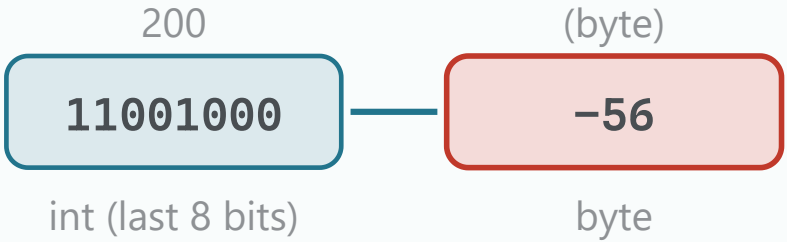
VARIABLES

`d1 = 2.0` `d2 = 2.25` `d3 = 2.0` `k = 9` `r = 10`

OUTPUT

Type Conversion: Tracing the Code

```
1 int a = 9, b = 4;
2 double d1 = a / b;
3 double d2 = (double) a / b;
4 double d3 = (double) (a / b);
5 int k = (int) 9.75;
6 long r = Math.round(9.75);
▶ byte y = (byte) 200;
8 System.out.println(d1 + " " + d2 + " " + d3);
9 System.out.println(k + " " + r + " " + y);
```



200 does not fit in a `byte`: the last 8 bits `11001000` are kept; the sign bit is 1 → 200 − 256 = **−56**.

Step 7 / 9

VARIABLES

`d1 = 2.0` `d2 = 2.25` `d3 = 2.0` `k = 9` `r = 10` `y = −56`

OUTPUT

Type Conversion: Tracing the Code

```
1 int a = 9, b = 4;
2 double d1 = a / b;
3 double d2 = (double) a / b;
4 double d3 = (double) (a / b);
5 int k = (int) 9.75;
6 long r = Math.round(9.75);
7 byte y = (byte) 200;
▶ System.out.println(d1 + " " + d2 + " " + d3);
9 System.out.println(k + " " + r + " " + y);
```

d1	d2	d3	k	r	y
2.0	2.25	2.0	9	10	-56
double	double	double	int	long	byte

`d1`, `d2`, `d3` are printed: `2.0 2.25 2.0`.

Step 8 / 9

VARIABLES

`d1 = 2.0` `d2 = 2.25` `d3 = 2.0` `k = 9` `r = 10` `y = -56`

OUTPUT

```
2.0 2.25 2.0
```

Type Conversion: Tracing the Code

```
1 int a = 9, b = 4;
2 double d1 = a / b;
3 double d2 = (double) a / b;
4 double d3 = (double) (a / b);
5 int k = (int) 9.75;
6 long r = Math.round(9.75);
7 byte y = (byte) 200;
8 System.out.println(d1 + " " + d2 + " " + d3);
▶ System.out.println(k + " " + r + " " + y);
```

d1	d2	d3	k	r	y
2.0	2.25	2.0	9	10	−56
double	double	double	int	long	byte

`k`, `r`, `y` are printed: `9 10 −56`.

Step 9 / 9

VARIABLES

`d1 = 2.0` `d2 = 2.25` `d3 = 2.0` `k = 9` `r = 10` `y = −56`

OUTPUT

```
2.0 2.25 2.0
9 10 -56
```

char ↔ int Conversion

- Every `char` is really a number: the character's Unicode code. `'A'` = 65, `'B'` = 66, `'a'` = 97.
- `char` → `int` is implicit; `int` → `char` needs `(char)`.
- Adding two `char` values with `+` gives a **number**; to get text, start with a `String`.
- Uppercase and lowercase English letters differ by 32 in code; digit codes start at 48 and are consecutive, so `'7' - '0'` gives the digit's numeric value (7).

For all character codes, see an ASCII table.

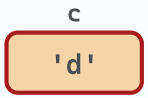
```
char ch = 'A';
int code = ch;                // 65
System.out.println(code);
char next = (char) (code + 1);
System.out.println(next);
System.out.println('B' + 'C');
System.out.println("" + 'B' + 'C');
System.out.println((char) (ch + 32));
System.out.println((int) '$');
```

OUTPUT

```
65
B
133
BC
a
351
```

char ↔ int: Tracing the Code

```
▶ char c = 'd';
2 int code = c;
3 char upper = (char) (c - 32);
4 int diff = 'z' - c;
5 char next = c;
6 next++;
7 System.out.println(code + " " + upper +
  " " + diff + " " + next);
8 System.out.println(c + 1);
9 System.out.println("" + c + 1);
```



`c` is a `char`: it stores the Unicode code of `'d'`, **100**.

Step 1 / 8

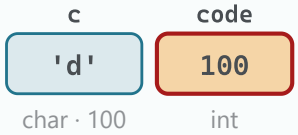
VARIABLES

`c = 'd'`

OUTPUT

char ↔ int: Tracing the Code

```
1 char c = 'd';
▶ int code = c;
3 char upper = (char) (c - 32);
4 int diff = 'z' - c;
5 char next = c;
6 next++;
7 System.out.println(code + " " + upper +
  " " + diff + " " + next);
8 System.out.println(c + 1);
9 System.out.println("" + c + 1);
```



`char` → `int` is an implicit conversion: `code = 100`. No cast needed.

Step 2 / 8

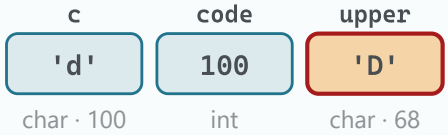
VARIABLES

`c = 'd'` `code = 100`

OUTPUT

char ↔ int: Tracing the Code

```
1 char c = 'd';
2 int code = c;
▶ char upper = (char) (c - 32);
4 int diff = 'z' - c;
5 char next = c;
6 next++;
7 System.out.println(code + " " + upper +
  " " + diff + " " + next);
8 System.out.println(c + 1);
9 System.out.println("" + c + 1);
```



`c - 32` is an `int` expression: 100 − 32 = 68. Going from `int` to `char` requires `(char)`: 68 = `'D'`.

Step 3 / 8

VARIABLES

`c = 'd'` `code = 100` `upper = 'D'`

OUTPUT

char ↔ int: Tracing the Code

```
1 char c = 'd';
2 int code = c;
3 char upper = (char) (c - 32);
▶ int diff = 'z' - c;
5 char next = c;
6 next++;
7 System.out.println(code + " " + upper +
  " " + diff + " " + next);
8 System.out.println(c + 1);
9 System.out.println("" + c + 1);
```



Subtracting two `char` values gives an `int`: `'z' - 'd'` = 122 − 100 = 22.

Step 4 / 8

VARIABLES

`c = 'd'` `code = 100` `upper = 'D'` `diff = 22`

OUTPUT

char ↔ int: Tracing the Code

```
1 char c = 'd';
2 int code = c;
3 char upper = (char) (c - 32);
4 int diff = 'z' - c;
▶ char next = c;
▶ next++;
7 System.out.println(code + " " + upper +
  " " + diff + " " + next);
8 System.out.println(c + 1);
9 System.out.println("" + c + 1);
```

c	code	upper	diff	next
'd'	100	'D'	22	'e'
char · 100	int	char · 68	int	char · 101

`next` starts as `'d'`; `next++` makes it 101 = `'e'`. `++` on a `char` variable needs no cast.

Step 5 / 8

VARIABLES

`c = 'd'` `code = 100` `upper = 'D'` `diff = 22` `next = 'e'`

OUTPUT

char ↔ int: Tracing the Code

```
1 char c = 'd';
2 int code = c;
3 char upper = (char) (c - 32);
4 int diff = 'z' - c;
5 char next = c;
6 next++;
▶ System.out.println(code + " " + upper +
  " " + diff + " " + next);
8 System.out.println(c + 1);
9 System.out.println("" + c + 1);
```

c	code	upper	diff	next
'd'	100	'D'	22	'e'
char · 100	int	char · 68	int	char · 101

Because of the `" "` string, the values are concatenated as text: `100 D 22 e`.

Step 6 / 8

VARIABLES

`c = 'd'` `code = 100` `upper = 'D'` `diff = 22` `next = 'e'`

OUTPUT

```
100 D 22 e
```

char ↔ int: Tracing the Code

```
1 char c = 'd';
2 int code = c;
3 char upper = (char) (c - 32);
4 int diff = 'z' - c;
5 char next = c;
6 next++;
7 System.out.println(code + " " + upper +
  " " + diff + " " + next);
▶ System.out.println(c + 1);
9 System.out.println("" + c + 1);
```

c	code	upper	diff	next
'd'	100	'D'	22	'e'
char · 100	int	char · 68	int	char · 101

`c + 1 → 100 + 1 = 101 (int)`

`c + 1` : `char` is promoted to a number; 100 + 1 = **101** is printed (not a letter).

Step 7 / 8

VARIABLES

`c = 'd'` `code = 100` `upper = 'D'` `diff = 22` `next = 'e'`

OUTPUT

```
100 D 22 e
101
```

char ↔ int: Tracing the Code

```
1 char c = 'd';
2 int code = c;
3 char upper = (char) (c - 32);
4 int diff = 'z' - c;
5 char next = c;
6 next++;
7 System.out.println(code + " " + upper +
  " " + diff + " " + next);
8 System.out.println(c + 1);
▶ System.out.println("" + c + 1);
```

c	code	upper	diff	next
'd'	100	'D'	22	'e'
char · 100	int	char · 68	int	char · 101

`"" + c + 1 → "d" + 1 → "d1"`

`"" + c` first gives the string `"d"`; then `+ 1` is appended → **d1**.

Step 8 / 8

VARIABLES

`c = 'd'` `code = 100` `upper = 'D'` `diff = 22` `next = 'e'`

OUTPUT

```
100 D 22 e
101
d1
```

Arithmetic Operators

Operator	Meaning	`x = 17, y = 5`	Result
`+`	addition	`x + y`	22
`-`	subtraction	`x - y`	12
`*`	multiplication	`x * y`	85
`/`	division	`x / y`	3
`%`	remainder (mod)	`x % y`	2
		`-17 / 5`	−3
		`-17 % 5`	−2
		`17.0 / 5`	3.4

- `/` on two integers does **integer division**.
- `%` (mod) gives the remainder, signed like the dividend.
- Even test: `n % 2 == 0`.
- Last digit: `n % 10`; drop it: `n / 10`.
- No power operator (`2 ** 3` fails): `Math.pow(2, 3)` → `8.0` (`double`).



`17 / 5 = 3 (full groups)` **`17 % 5 = 2`**

3 groups of five, 2 left over

`+` with a `String` operand **concatenates**: `"Age: " + 20` → `Age: 20`.

Increment (++) and Decrement (--): Trace

`x++` and `++x` both increase `x` by 1; the difference is in the expression's **value**: postfix gives the old value, prefix the new one. As a standalone `i++;` statement there is no difference.

```
▶ int a = 5;
2 int b = a++;
3 System.out.println(a + " " + b);
4 int c = ++a;
5 System.out.println(a + " " + c);
6 int d = a--;
7 System.out.println(a + " " + d);
```



`a` starts at 5.

Step 1 / 7

VARIABLES

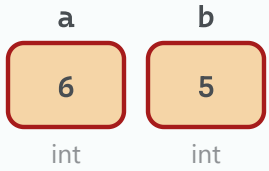
`a = 5`

OUTPUT

Increment (++) and Decrement (--): Trace

`x++` and `++x` both increase `x` by 1; the difference is in the expression's **value**: postfix gives the old value, prefix the new one. As a standalone `i++;` statement there is no difference.

```
1 int a = 5;
▶ int b = a++;
3 System.out.println(a + " " + b);
4 int c = ++a;
5 System.out.println(a + " " + c);
6 int d = a--;
7 System.out.println(a + " " + d);
```



Postfix `a++` : the old value (5) is used first → `b = 5` , **then** `a` becomes 6.

Step 2 / 7

VARIABLES

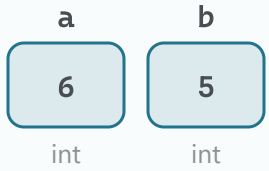
`a = 6` `b = 5`

OUTPUT

Increment (++) and Decrement (--): Trace

`x++` and `++x` both increase `x` by 1; the difference is in the expression's **value**: postfix gives the old value, prefix the new one. As a standalone `i++;` statement there is no difference.

```
1 int a = 5;
2 int b = a++;
▶ System.out.println(a + " " + b);
4 int c = ++a;
5 System.out.println(a + " " + c);
6 int d = a--;
7 System.out.println(a + " " + d);
```



Printed: `6 5`.

Step 3 / 7

VARIABLES

`a = 6` `b = 5`

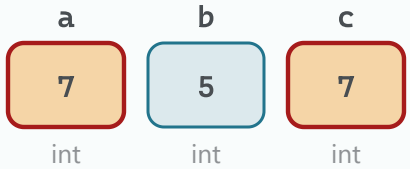
OUTPUT

```
6 5
```

Increment (++) and Decrement (--): Trace

`x++` and `++x` both increase `x` by 1; the difference is in the expression's **value**: postfix gives the old value, prefix the new one. As a standalone `i++;` statement there is no difference.

```
1 int a = 5;
2 int b = a++;
3 System.out.println(a + " " + b);
▶ int c = ++a;
5 System.out.println(a + " " + c);
6 int d = a--;
7 System.out.println(a + " " + d);
```



Prefix `++a`: **first** `a` becomes 7, then the new value is used → `c = 7`.

Step 4 / 7

VARIABLES

`a = 7` `b = 5` `c = 7`

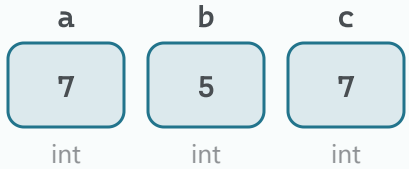
OUTPUT

```
6 5
```

Increment (++) and Decrement (--): Trace

`x++` and `++x` both increase `x` by 1; the difference is in the expression's **value**: postfix gives the old value, prefix the new one. As a standalone `i++;` statement there is no difference.

```
1 int a = 5;
2 int b = a++;
3 System.out.println(a + " " + b);
4 int c = ++a;
▶ System.out.println(a + " " + c);
6 int d = a--;
7 System.out.println(a + " " + d);
```



Printed: `7 7`.

Step 5 / 7

VARIABLES

`a = 7` `b = 5` `c = 7`

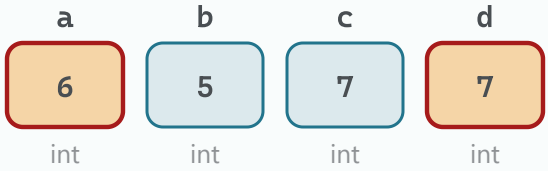
OUTPUT

```
6 5
7 7
```

Increment (++) and Decrement (--): Trace

`x++` and `++x` both increase `x` by 1; the difference is in the expression's **value**: postfix gives the old value, prefix the new one. As a standalone `i++;` statement there is no difference.

```
1 int a = 5;
2 int b = a++;
3 System.out.println(a + " " + b);
4 int c = ++a;
5 System.out.println(a + " " + c);
▶ int d = a--;
7 System.out.println(a + " " + d);
```



`a--` : the old value (7) first → `d = 7`, then `a` becomes 6.

Step 6 / 7

VARIABLES

`a = 6` `b = 5` `c = 7` `d = 7`

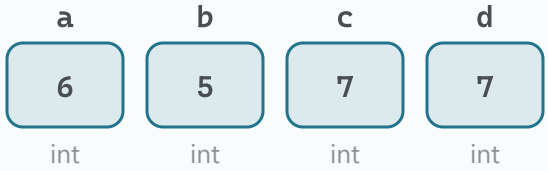
OUTPUT

```
6 5
7 7
```

Increment (++) and Decrement (--): Trace

`x++` and `++x` both increase `x` by 1; the difference is in the expression's **value**: postfix gives the old value, prefix the new one. As a standalone `i++;` statement there is no difference.

```
1 int a = 5;
2 int b = a++;
3 System.out.println(a + " " + b);
4 int c = ++a;
5 System.out.println(a + " " + c);
6 int d = a--;
▶ System.out.println(a + " " + d);
```



Printed: `6 7`.

Step 7 / 7

VARIABLES

`a = 6` `b = 5` `c = 7` `d = 7`

OUTPUT

```
6 5
7 7
6 7
```

Operators and Precedence

Precedence	Operators	Associativity
1 (highest)	postfix `x++` `x--`	→
2	unary: `+x` `-x` `!` `++x` `--x` · cast `(int)`	←
3	multiplicative: `*` `/` `%`	→
4	additive: `+` `-`	→
5	relational: `<` `>` `<=` `>=`	→
6	equality: `==` `!=`	→
7	logical AND: `&&`	→
8	logical OR: `||`	→
9	ternary: `? :`	←
10 (lowest)	assignment: `=` `+=` `-=` `*=` …	←

At equal precedence, **binary** operators group left to right (→): `10 - 4 - 3` = 3. **Unary** operators and assignment group right to left (←): `a = b = 0`. To change the order or improve readability, use **parentheses**.

Evaluating an Expression Step by Step

```
System.out.println(2 + 3 * 4 - 6 / 2);
```

2 + 3 * 4 - 6 / 2

`*` and `/` come before `+` and `-`; of the two, the **leftmost** goes first: 3 * 4 = 12.

Step 1 / 5

`(2 + 3) * 4`
= 5 * 4 = **20**

`10 - 4 - 3`
= 6 - 3 = **3** (left to right)

`20 / 4 * 2`
= 5 * 2 = **10** (left to right)

`-2 * -3`
unary minus first: **6**

Evaluating an Expression Step by Step

```
System.out.println(2 + 3 * 4 - 6 / 2);
```

2 + 12 - 6 / 2

Next highest-precedence operation: 6 / 2 = 3.

Step 2 / 5

`(2 + 3) * 4`
= 5 * 4 = **20**

`10 - 4 - 3`
= 6 - 3 = **3** (left to right)

`20 / 4 * 2`
= 5 * 2 = **10** (left to right)

`-2 * -3`
unary minus first: **6**

Evaluating an Expression Step by Step

```
System.out.println(2 + 3 * 4 - 6 / 2);
```

2 + 12 − 3

The remaining `+` and `-` have equal precedence → left to right: 2 + 12 = 14.

`(2 + 3) * 4`
= 5 * 4 = **20**

`10 - 4 - 3`
= 6 - 3 = **3** (left to right)

`20 / 4 * 2`
= 5 * 2 = **10** (left to right)

`-2 * -3`
unary minus first: **6**

Evaluating an Expression Step by Step

```
System.out.println(2 + 3 * 4 - 6 / 2);
```

`14 - 3`

14 − 3 = 11.

Step 4 / 5

`(2 + 3) * 4`
= 5 * 4 = **20**

`10 - 4 - 3`
= 6 - 3 = **3** (left to right)

`20 / 4 * 2`
= 5 * 2 = **10** (left to right)

`-2 * -3`
unary minus first: **6**

Evaluating an Expression Step by Step

```
System.out.println(2 + 3 * 4 - 6 / 2);
```

11

Result **11**. Parentheses change the order: `(2 + 3) * 4 - 6 / 2` = 20 − 3 = 17.

Step 5 / 5

`(2 + 3) * 4`
= 5 * 4 = **20**

`10 - 4 - 3`
= 6 - 3 = **3** (left to right)

`20 / 4 * 2`
= 5 * 2 = **10** (left to right)

`-2 * -3`
unary minus first: **6**

Comparison (Relational) Operators

Operator	Meaning	`x = 10, y = 20`
`==`	equal?	`x == y` → `false`
`!=`	not equal?	`x != y` → `true`
`>`	left greater?	`y > x` → `true`
`<`	left smaller?	`x < y` → `true`
`>=`	greater or equal?	`x >= 30` → `false`
`<=`	less or equal?	`x <= y` → `true`

- Evaluates the relationship between values; the result is always a `boolean`: `true` or `false`.
- The result can be stored in a variable: `boolean isGreater = y > x;`
- `=` is assignment, `==` is comparison. `if (x = 5)` does not compile:

```
error: incompatible types: int cannot be
       converted to boolean
```

Logical Operators

- **AND** `&&` : if both conditions are `true` , the result is `true` . `A && B` is true only when both A and B are true.
- **OR** `||` : if at least one is `true` , the result is `true` .
- **NOT** `!` : flips the value: `!true` → `false` .

A	B	A && B	A || B	!A
true	true	**true**	**true**	false
true	false	false	**true**	false
false	true	false	**true**	**true**
false	false	false	false	**true**

```
int x = 10, y = 20, z = 30;
System.out.println(x < y && y < z);
System.out.println(x > y || y < z);
System.out.println(!(x < y));
int age = 20;
boolean hasLicense = true;
System.out.println(age >= 18 && hasLicense);
```

OUTPUT

```
true
true
false
true
```

Short-Circuit Evaluation

- `&&` : if the left side is `false`, the result is already `false` : the right side is **never evaluated**.
- `||` : if the left side is `true`, the right side is skipped: `day == 6 || day == 7` — with day = 6, the second comparison is never made. Its negation `!(day == 6 || day == 7)` means "weekday".
- This guards a dangerous operation: first check "is the divisor nonzero?", and divide only if it is.
- Order matters: had you written `num / divisor > 1 && divisor != 0`, the program would stop with a division-by-zero error (`ArithmeticException`).

```
int divisor = 0;
int num = 10;
if (divisor != 0 && num / divisor > 1) {
    System.out.println("Ratio above 1");
} else {
    System.out.println("Can't divide or too small");
}
```

OUTPUT

```
Can't divide or too small
```

OPERATORS

The Ternary Operator

A short expression that picks one of two values by a condition: if the condition is true, `value1`, otherwise `value2`.

```
result = (condition) ? value1 : value2;
```

Example: is a number positive or negative?

```
int num = -5;
String result = (num > 0) ? "Positive" : "Negative";
System.out.println("Result: " + result);
```

OUTPUT

```
Result: Negative
```

The same with `if-else`:

```
String result;
if (num > 0) {
    result = "Positive";
} else {
    result = "Negative";
}
```

`?:` produces a **value**, so it can go on the right of an assignment or inside `println`. Both options must have compatible types: `double abs = (num < 0) ? -num : num;` (−4.5 → 4.5), `int max = (a > b) ? a : b;` (8, 13 → 13).

Nested Ternary Operators

Ternary operators can be nested to express more complex conditions briefly. Example: is a number zero, positive or negative?

```
int num = -5;
String result = (num == 0) ? "Zero" : (num > 0) ? "Positive" : "Negative";
System.out.println("Result: " + result);
```

num	Result
0	`Zero`
7	`Positive`
−5	`Negative`

Read it as: “zero? yes → Zero; no → positive? yes → Positive; no → Negative”. The second `?:` is the first one's `value2` (they group from the right).

More than two nested `?:` hurt readability; use `if-else if` instead.

CONTROL FLOW

Syntax of Control Statements

Statement	Syntax	Rule
`if` / `else`	`if (condition) { … } else { … }`	the condition must be `boolean`; `if (…);` : the trailing `;` is an empty statement
`switch`	`switch (d) { case 1: … break; default: … }`	`int`, `char`, `String`, `enum`; `break` missing → falls through to the cases below
`switch` (Java 14+)	`x = switch (d) { case 1 -> "P"; default -> "?"; };`	no fall-through; an expression that yields a value
`for`	`for (int i = 0; i < n; i++) { … }`	init once; condition before each pass, update after
`while`	`while (condition) { … }`	body may never run
`do-while`	`do { … } while (condition);`	body runs at least once; the trailing `;` is required
`break` / `continue`	`break;` · `continue;`	`break` exits, `continue` skips the rest of the pass

Every `for` can be rewritten as a `while`. A loop with no step that makes its condition false is an infinite loop.

Summary

Topic	Key points
Compilation	`javac Name.java` → `Name.class` (bytecode) · `java Name` → the JVM runs it; the same bytecode on every platform
Syntax	statements end with `;` · blocks `{ }` · case-sensitive · `//` and `/* */` comments
Primitive types	`byte` 1, `short` 2, `int` 4, `long` 8 · `float` 4, `double` 8 · `char` 2 bytes · `boolean` ; values past the limit overflow
Names	letters, digits, `_` , `$` ; not starting with a digit; not a keyword · classes PascalCase, variables camelCase, CONSTANT_NAME
Assignment	right side evaluated first · `+=` … shorthands · local variables must be assigned before being read · `final` constants
Conversion	`byte → short → int → long → float → double` automatic; the reverse needs `(type)` , fraction dropped · `int / int` integer division
Operators	unary > `* / %` > `+ −` > comparison > `&&` > `||` > `?:` > assignment · `&&` `||` short-circuit
Flow	`if/else` , `switch` (+ `break`), `?:` · `for` , `while` (0+ passes), `do-while` (1+ passes) · `break` exits, `continue` skips

SUMMARY

Common Mistakes (1)

Missing semicolon

`error: ';' expected`. The error is usually at the end of the **previous** line.

File name ≠ class name

`class HelloWorld is public, should be declared in a file named HelloWorld.java`

Upper/lower case

`string`, `Int` do not compile: `cannot find symbol`; `system.out` gives `package system does not exist`. Correct: `System`, `String`, `int`.

Uninitialized local variable

`variable count might not have been initialized`: local variables get no default value.

Literal without a suffix

`float f = 3.14;` → `possible lossy conversion from double to float` (use `3.14f`) · `long l = 3000000000;` → `integer number too large` (`L` suffix needed).

Forgetting the cast when narrowing

`int → char`, `double → int` assignments give `incompatible types: possible lossy conversion`; you need `(char)`, `(int)`.

SUMMARY

Common Mistakes (2): Compiles but Wrong

Integer division

`double avg = (3 + 4) / 2;` → **3.0**. Correct: `(3 + 4) / 2.0` → 3.5.

`=` vs. `==`

`if (x = 5)` → `int cannot be converted to boolean`. Comparison uses `==`.

`switch` with no `break`

Every case after the matching one runs too (fall-through).

`if` followed by `;`

`if (x > 0); { … }`: the empty statement becomes the body, and the block **always** runs.

Infinite loop

`for (int i = 1; i <= 10; i--)`, or forgetting to update the condition variable in a `while`.

Floating-point `==`

`0.1 + 0.2 == 0.3` → `false`. Check that the difference is tiny: `Math.abs(a - b) < 1e-9`.

SUMMARY

Practice Problems

What does each line print? (`int a = 7, b = 2;`)

1. `System.out.println(a / b + " " + a % b + " " + (double) a / b);`
2. `System.out.println(1 + 2 + "3" + 4 + 5);`
3. `int i = 3; int j = i++ * 2;` → `i` and `j` = ?
4. `byte k = (byte) 200;` → `k` = ?
5. `for (int m = 10; m > 0; m -= 3) System.out.print(m + " ");`
6. `System.out.println('a' + 1);` and `System.out.println((char) ('a' + 1));`

▶ **Show answers**

SUMMARY

Exercises

1. Convert Celsius to Fahrenheit: F = C × 9 / 5 + 32. What is the result for C = 37? With `int` arithmetic, what happens if you write `9 / 5 * c + 32`?
2. Split 3725 seconds into hours, minutes and seconds with `/` and `%`.
3. Print whether a number is odd or even **using the ternary operator**.
4. Add up the numbers from 1 to 100 that are **not** multiples of 3, using `continue`.
5. `char letter = 'g';` — convert this character to uppercase using the code difference.
6. Which of these are valid identifiers: `_total`, `3size`, `studentNo`, `final`, `full name`, `Final`?

▶ **Answers / hints**

References

- Oracle, *The Java Tutorials*: "Language Basics" (variables, operators, expressions, control flow), docs.oracle.com/javase/tutorial.
- J. Gosling et al., *The Java Language Specification*, Java SE 21 Edition (Chapter 3: lexical structure, keywords, literals; Chapter 4: types; Chapter 5: conversions).
- T. Lindholm et al., *The Java Virtual Machine Specification*, Java SE 21 Edition (bytecode and the JVM).
- P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson, 2017 (Chapters 2–5).
- K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly, 2022.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 13th ed., 2024 (Chapter 3).
- A. B. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019.

The End

Chapter 2.2: Syntax Rules

Previous: Chapter 2.1: Programming · Next: Chapter 3.1: Control Structures